Towards a Discipline of Performance Engineering: Lessons Learned from Stencil Kernel Benchmarks

Inauguraldissertation

zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Danilo Guerrera

aus Italien

Basel, 2019

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Em. Dr. Helmar Burkhart
Prof. Dr. Gerhard Wellein

Basel, den 13. November 2018

Prof. Dr. Martin Spiess,
Dekan

To my family, who encouraged me to pursue my dreams and finish my dissertation.

Abstract

High performance computing systems are characterized by a high level of complexity both on their hardware and software side. The hardware has evolved offering a lot of compute power, at the cost of an increasing effort needed to program the systems, whose software stack can be correctly managed only by means of ad-hoc tools.

Reproducibility has always been one of the cornerstones of science, but it is highly challenged by the complex ecosystem of software packages that run on HPC platforms, and also by some malpractices in the description of the configurations adopted in the experiments.

In this work, we first characterize the factor that affects the reproducibility of experiments in the field of high performance computing and then we define a taxonomy of the experiments and levels of reproducibility that can be achieved, following the guidelines of a framework that is presented.

A tool that implements said framework is described and used to conduct Performance Engineering experiments on kernels containing the stencil (structured grids) computational pattern. Due to the trends in architectural complexity of the new compute systems and the complexity of the software that runs on them, the gap between expected and achieved performance is widening. Performance engineering is critical to address such a gap, with its cycle of prediction, reproducible measurement and optimization.

A selection of stencil kernels is first modeled and their performance predicted through a grey box analysis and then compared against the reproducible measurements. The prediction is then used to validate the measured performance and vice-versa, resulting in a "Gold Standard" that draws a path towards a discipline of performance engineering.

Contents

Contents i

1 Introduction 3
- 1.1 Research Questions . . . 6
- 1.2 Related Work . . . 6
- 1.3 Organization of This Work . . . 11

I Reproducibility Challenges 13

2 Reproducibility Challenges: Hardware Complexity 15
- 2.1 Moore's Law . . . 15
- 2.2 From Single Core to Multicore, Manycore, and Accelerators 19
  - 2.2.1 Pipelining . . . 21
  - 2.2.2 Out-of-Order Execution . . . 23
  - 2.2.3 SIMD . . . 23
  - 2.2.4 Superscalarity . . . 25
  - 2.2.5 Parallel Computers . . . 25
  - 2.2.6 Simultaneous Multi-Threading . . . 26
  - 2.2.7 Dynamic Frequency Scaling . . . 27
- 2.3 Memory Subsystem . . . 27
  - 2.3.1 Shared Memory Computers . . . 28
  - 2.3.2 Real Life Intermezzo . . . 32
- 2.4 Network Subsystem . . . 33

3 Reproducibility Challenges: Software Complexity 35
- 3.1 Amdahl's and Gustafson's Laws . . . 35
- 3.2 Programming Models . . . 38

3.3 Compilers . . . 39
3.3.1 Basic Optimizations . . . 39
3.3.2 Loop Optimizations . . . 40
3.4 Software Stack . . . 42
3.4.1 Environment . . . 42
3.4.2 Build Process . . . 43

II The PROVA! Approach to Reproducible HPC Research 45

4 Proposed Framework 47
4.1 Taxonomy of Experiments . . . 50
4.2 Reproducibility Levels . . . 51
4.3 Goals of the PROVA! Project . . . 52
4.3.1 Contributions to the Project . . . 54

5 Overview of the PROVA! Tool 55
5.1 Addressing Complexity . . . 56
5.1.1 Implementation Phase . . . 57
5.1.2 Compilation Phase . . . 57
5.1.3 Execution Phase . . . 57
5.2 Walkthrough . . . 58

6 Implementation Aspects 63
6.1 Architecture . . . 63
6.2 Mapping of the Experiment Taxonomy . . . 66
6.2.1 Projects . . . 66
6.2.2 Methods . . . 67
6.3 Likwid Interface . . . 67
6.4 Empirical Roofline . . . 70

III Experimental Evaluation 73

7 Parallel Stencil Codes 75
7.1 Motifs . . . 75
7.2 Stencils Classification . . . 77
7.3 Stencil TEMPlating Engineering Library . . . 80
7.4 Stencil Compilers . . . 82

7.4.1 PLUTO . . . 82
7.4.2 PATUS . . . 83

8 Performance Evaluation 85
8.1 Performance Analysis . . . 85
8.1.1 Performance Models . . . 86
8.1.2 Grey Box Modeling . . . 90
8.2 Performance Measurement . . . 92
8.2.1 Cache Misses . . . 93
8.2.2 Code Structure and Parallelism . . . 94
8.2.3 Memory Access Pattern and Data Locality . . . 94
8.2.4 Optimization of Stencil Codes . . . 96

9 Experimental Testbeds 99
9.1 Systems . . . 100
9.1.1 Validation Macro-Experiment . . . 100
9.1.2 Performance Engineering Cycle Macro-Experiment 100
9.1.3 Emmy . . . 101
9.1.4 MiniHPC . . . 101
9.2 Problems . . . 107
9.2.1 Kernels Used in the Validation Macro-Experiment . 107
9.2.2 Kernels Used in the Performance Engineering Cycle 108
9.3 Methods . . . 109
9.3.1 Methods Used in the Validation Macro-Experiment 109
9.3.2 Methods Used in the Performance Engineering Cycle Macro-Experiment . . . 109

10 Performance Benchmarking Experiments 111
10.1 Stencil Compilers . . . 111
10.1.1 Metric used for Compilers Evaluation and Comparison . . . 112
10.2 Discussion of the Results . . . 121
10.2.1 Validation Experiment . . . 121
10.2.2 Performance Engineering Cycle Experiment . . . 123

IV Conclusions & Future Work 133

11 Conclusions and Future Work 135
11.1 Contributions and Relevance to the Community . . . 137

11.2 Future Work 138

Bibliography **141**

Appendices **155**

A Stencils Source Code **157**
A.1 2d-1r-iso-const-box 157
A.1.1 PATUS 157
A.1.2 PLUTO 158
A.1.3 OpenMP 159
A.2 2d-4r-iso-const-box 160
A.2.1 PATUS 160
A.2.2 PLUTO 162
A.2.3 OpenMP 164

B Creation of a MethodType **167**

C Walkthrough of STEMPEL: Kerncraft and PROVA! Interfaces **171**

List of Figures **175**

List of Tables **181**

Listings **182**

Acknowledgments

The Ph.D. is a major phase of the academic life. I am very glad and thankful to Prof. Em. Dr. Helmar Burkhart for having given me the opportunity to work with him on this project, for his guidance and for sharing with me and the whole High Performance and Web Computing (HPWC) group his passion for research.

I would like to thank also Prof. Dr. Gerhard Wellein, who hosted me in his group at the Friedrich-Alexander-Universität Erlangen-Nürnberg as a temporary researcher, and for accepting to act as co-referee in the thesis committee. He has taught me a lot regarding performance engineering.

I wish to thank all the members of the (wide) HPWC group: Dr. Martin Guggisberg, for his optimism and valuable ideas and suggestions, Alexander Gröflin who always had a joke ready for every situation, Robert Frank for the stimulating discussions, Bas Kin for his support (mostly on the pitch), Dominic Bosch for the inspiring conversations, Yvonne Wegmüller who helped to solve all the administrative issues I ever faced. A special thanks goes to Antonio Maffia, who shared with me a long path: we started the bachelor together at the University of Sannio, in Italy, shared a lot of good and also hard moments in this run until the completion of the Ph.D process. Thanks for your help all over these years.

I would like to express my gratitude to Prof. Dr. Florina Ciorba, for accepting me in her group and always motivating me, and to all the members of the HPC group: Ali Mohammed, Aurélien Cavelan, Ahmed Eleliemy, and Jonas Korndörfer.

I am grateful for having met and worked with: Alexander Ugolini, Dr. Georg Hager, Dr. Jan Eitzinger, Thomas Röl, and Julian Hammer.

Finally, I would like to thank my parents and my brother, for their constant support, even though during the time of the Ph.D. they have been far away from me. I hope you can all be proud of me.

Thanks to my aunts and uncles Maria, Angelo, Antonietta, Anna, Luciano, and my cousins Sara, Joele, and Mattia, who love me and motivate me anytime and anyway they can.

Thanks to Winnie, for staying by my side also in the awkward moments that the path I walk sometimes makes me face, I love you.

Grazie mamma e babbo, vi voglio bene!

Chapter 1

Introduction

Modern architectures are characterized by a high level of complexity, due to the limits the producers have been facing by 2007 when the natural scaling of the performance was provided for free by the scaling in the frequency of the chips. Said limits forced the vendors to find a different way to deliver performance, i.e., alternative circuits and usage of the transistors: the switch between single core and multicore happened. The architectures have been developing following this trend since then: accelerators and many-core architectures also came out. To properly exploit these architectures, the parallelism must be present in the source code of the applications, at a fine grain. Developers are required to know extremely good how the microprocessors work. This way they can tune their codes and exploit the architectural features of their machines. This kind of optimization is extremely error-prone, and performance portability can suffer out of it.

State of the art in performance reporting in the High Performance Computing (HPC) field is omitting details that are important to be able to test and reuse the proposed approaches, affecting what is considered to be a pillar of science since the 17th century: the scientific method [136]. Every scientist must be able to understand and extend the work of another.

In the HPC field, we refer to the computing environment and its setup both in terms of hardware and software configurations. Modern architectures are incredibly complex, and scientists often focus their efforts on what they are pursuing, ignoring the importance of making their science reproducible. We acknowledge the lack of time and the effort nec-

essary to follow good practices in conducting/doing research in computational science, yet consider reproducibility to be extremely important. We present a taxonomy of reproducibility: it is tricky to use the terms consistently in this field since there is no universal agreement on them [83]. Recently ACM defined the 3 Rs of the Research, and their description is mostly overlapping our taxonomy. This work takes over such discussion and proposes three levels of reproducibility with an explanation of what they mean and what they provide if research fulfills their constraints. Moreover, we propose a standardized way of describing an experiment, which extrapolates a minimum amount of information needed for reproducibility purposes and a workflow to follow in order to carry out an experiment and achieve reproducible results properly. Starting from our experience and the proposed approach to reproducible research, we implemented PROVA!, a distributed workflow and system management tool, that helps scientists in their study allowing them to focus on their core business and taking care of its reproducibility, and then used it in our own research on benchmarking of parallel stencil codes. Stencil codes are an essential and widely used pattern in computational science. The stencil computational pattern is representative of many numerical codes, from PDE to multi-grid solvers, and discrete simulations, including image filtering. Usually, the stencil computation represents a significant part of the overall execution time of an application. Therefore, it is crucial to achieving good performance in calculating them. Because of their low arithmetic intensity, i.e., the small number of floating point operations per transferred data element, stencil computations typically are limited by the available bandwidth to the memory subsystem. Thus, it is essential to efficiently use the caches, by optimizing data locality (both spatial and temporal). Bandwidth-saving schemes like cache blocking techniques and methods to block across multiple time steps are used to increase the arithmetic intensity. Hardware-aware programming techniques can also help: NUMA-aware data initialization, software prefetching, or cache bypassing represent a way to reduce bandwidth usage further. All the listed concepts make the optimization of stencil codes a tough challenge. Lately, several approaches to stencil computation have been proposed, and several stencil compilers have been implemented.

Focusing on the problem of stencil computation we consider both manually optimized codes and stencil compilers. When comparing the results obtained we must acknowledge that there are many factors involved that can affect the performance of a code, starting from the stencil

itself. Real stencil applications span in a rich domain of characteristics, 2-dimensional vs. 3-dimensional, high vs. low order, dense vs. sparse matrices, aligned vs. staggered grids, widely varying in arithmetic intensity. Another factor involved is the system used for running a code: both the different hardware and different software environment, e.g., a different compiler or set of flags. Performance models are, thus, required to evaluate the expected performance of a code on a specific architecture, thus highlighting the parameters that can affect its performance, i.e., expose the bottlenecks that limit the performance of a code on a machine. For this purpose, we plan to interpret the performance data employing models such as the Execution-Cache-Memory (ECM) and the Roofline Model, evaluate how performance is affected by explicitly pinning the threads, following different pinning strategies (core, socket, node). The goal is to verify that modeling and execution match and to understand how compilers and manually optimized code behave concerning a specific architecture. An important aspect resides into the solution used to evaluate them while changing both the stencil to compute and the system: usually the developers of a stencil compiler focus either on a particular architecture or a particular stencil (or class of stencil computations). Using PROVA! it is possible to achieve a fair and reproducible comparison of the same stencil on different systems, or different stencils on both the same and different systems. Nonetheless, it is needed to provide a clear and standardized way of describing an architecture, to carry on comparisons. Studying the performance of a code, verifying it empirically, while taking care of its reproducibility, strengthen and gives credibility to research: the proposed framework aims to ease the road towards a discipline of performance engineering. Furthermore, making available the source code is a minimum pre-requisite for reproducibility, i.e., it is necessary but not sufficient. It is crucial to have a detailed description and provision of: dependencies, automated build process, environment, execution scripts post-processing scripts, and secondary data generating published figures.

Conducting research in such a way has several positive effects: first, the author of the study remains with complete documentation of the work. Additionally, it helps the follow-up of studies allowing to build upon existing work and, if there are discrepancies in the case of a replication attempt by another scientist, it helps in identifying and addressing the root of the problem.

1.1 Research Questions

- What are the factors that affect reproducibility in the context of performance benchmarking experiments and how is it possible to control them, thus sharing with the scientific community trusted and reproducible research?
- Performance models are needed to expose the bottlenecks of high performance architectures and to verify that the code implementations exploit the available hardware feature properly. How is it possible, to validate a performance model against a reproducible execution and vice-versa?
- Is it possible to fairly compare different approaches to the parallelization of stencil codes across different architectures?

1.2 Related Work

The Oxford English Dictionary [35] defines Reproducibility as "the extent to which consistent results are obtained when an experiment is repeated". It is usually expected a scientific experiment to be reproducible, but scientists have been bored in trying themselves to reproduce someone else's work or read about it. Journals do not accept the attempt of reproducing published findings driving to published science rarely be tested. As described in [24], "the assumption that science must be reproducible is implicit yet seldom tested, and in many systems, the exact reproducibility of experimental data is unknown or has not been rigorously investigated in a systematic fashion". The meaning of the terms reproducibility and replicability is highly discussed and questioned. In the biology field, scientists tend to believe that the reproducibility of an experiment coincides with the ability to replicate it. Drummond believes that these two terms are distinct and he argues that while reproducibility requires change, replicability avoids it [38]. In fact, according to Drummond, reproducibility is related to a phenomenon that can be predicted to recur even when there is a certain degree of variability in the experimental conditions, whereas replicability describes the ability to obtain an identical result when an experiment is performed under identical conditions. As stated in [28], "the principal goal of scientific publications is to teach new concepts, show the resulting implications of those concepts in an illustration, and provide enough detail to make the work re-

producible". Claerbout and Karrenbach define reproducibility in computational sciences as a nametag attached to every figure caption in a manuscript: said tag could be used to recalculate the figures from all the data, parameters and programs.

According to Liberman [83] it is inside the human nature to wonder whether or not an experiment would adequately work when performed by someone else than the original scientist, following a similar recipe. Between 1990 and 2006, people began to use terms like replication, replicable, replicability, to refer to the process of completely re-running an experiment, with all the permutations of new researchers, new equipment, new subjects or other raw materials. An outlook of the terminologies used for Reproducible Research is given by [13], trying to organize all the words used by different research groups or communities while referring to the broad topic of reproducibility.

Science gathers knowledge about the universe, organizes and merges it into testable laws and theories [82]. Its success and credibility are strictly related to the willingness of the scientists to test and replicate independently the work done by others. In order to do so, the complete exchange of procedures, materials, and data is needed. The desirability of reproducibility leads to the practical question of how many times an experiment should be reproduced before publication. By using the words of Karl Popper: "Non-reproducible single occurrences are of no significance to science". Reproducibility assures that the effect is not due to chance or an experimental artifact resulting in a one-time event.

The ability of a researcher to confirm an experimental result is essential to science, inherently assuming its reproducibility. It must be noted that there are practical limits to the reproducibility of findings. Although this question has not been formally studied, reproducibility is likely to be inversely proportional to the complexity of an experiment. The value and importance of reproducibility have been demonstrated by studies showing impossibility or difficulty to replicate published results [74], failed clinical trials [104] [14].

"It is impossible to believe most of the computational results presented at conferences and in published papers today. Even mature branches of science, despite all their efforts, suffer severely from the problem of errors in final published conclusions" [126]. Over the past few years, various researchers have made systematic attempts to reproduce some of the more widely cited priming experiments. Many of these attempts have failed. A recent paper reported that only a minority of published

microarray results could be repeated [74]. In 2011, Florian Prinz and his colleagues at Bayer HealthCare, a German pharmaceutical company, reported in Nature Reviews Drug Discovery that they had successfully reproduced the published results in just a quarter of 67 seminal studies [104]. Unfortunately, such findings are consistent with those of others scientists at Amgen, an American drug company, that tried to replicate 53 studies that they considered landmarks in the basic science of cancer, often co-operating closely with the original researchers to ensure that their experimental technique matched the one used in the original study. According to a piece they wrote in Nature [14], a leading scientific journal, they were able to reproduce the original results in just six cases. Such observations have even led some to question the validity of the requirement for reproducibility in science [133].

Experimental reproducibility remains a standard and accepted criterion for publication. As a consequence, researchers should endeavor to obtain information about the reproducibility of their works. The number of times that an experiment is performed should be clearly stated in a manuscript, and a new finding should preferably be reproduced more times. In general, an experiment that challenges existing assumptions is going to be inquired more cautiously than one matching the established models and paradigms. Thus, we can argue that reproducibility acquires greater value and importance proportionally to the importance of the new ideas or methodologies. The availability of new tools and technologies, the increased amount of data (Big Data Science), together with the development of more interdisciplinary approaches and thus the complexity of the questions being asked, contribute to making any replication effort more complicated [75]. In [100] Peng suggests to build up a minimum standard of reproducibility, believing that methods and code should be made available. Thus, other researchers may obtain consistent results and confirm the original work. Therefore, the papers should sufficiently describe not only the results but also the configuration of the experiment and the chain of steps that drove to specific outcomes, starting from the given input data [12, 94, 123, 125].

Nowadays researchers have the burden of publishing, catching up on the deadlines and do not have enough time to devolve into ensuring reproducibility. Even if a good practice with respect to the reproducibility may slow down research in the short term, it is necessary in order to allow previously developed methodologies to be effectively applied on new data, or to enable the reuse of code and results for new projects.

Additionally, it has been proven that sharing detailed research data is associated with an increased citation rate [102].

In the last years a movement promoting reproducible research through tools and best practices is emerging. The tools mostly target the reproducibility of the data analysis [124]. In the bioinformatics field, there are plenty of tools that make the researcher's life more comfortable and ensure reproducibility of their approach, such as Taverna [98], VisTRails [43, 117], Galaxy [50], GenePattern [108]. In our classification of reproducibility, we start from the basis, i.e., one must be able to ensure at least the reproducibility of his work.

Reference [115] presents ten simple rules for reproducibility of computational research. A researcher should keep always track of how a result was produced, avoiding manual manipulation of the data, while recording intermediate results in a standardized format: every detail that may influence the execution should be noted. In computational sciences, the critical details include names and versions of the programs used, as well as the parameters and their values. The format can range from a makefile to a shell script [118], to the complete workflow [50, 98, 52, 34]. Additionally one may want to use version control for the custom scripts or his code since even a slight modification to a computer program can alter its output significantly: backtracking to the code state that generated certain results may be an endless task. Version control systems like Git or Mercurial represent the state of the art solution for tracking the evolution of code. Last, all the information should be published and made publicly accessible. Most journals allow articles to be supplemented with online material.

Opening up access to the data and software, not just the final publication, is one of the goals of the open science movement, thus allowing to build upon existing work, either to test it or to develop new ideas. A correct sharing of data and code, together with a manuscript, may be beneficial. Consequently, scientists should share their artifacts, with licenses that if on one side protect the original author, on the other allow fair use of their work. Besides the ethical and social challenges, there is the practical one of how to give a comprehensive overview of one's work. Every scientist makes use of version control in his work: from the Software Engineering, we can leverage the long lasting solution of using Version Control Systems. A common feature of all types of VCS is the ability to save several versions of a file, together with a descriptive message, during the development phase. All the changes and messages are

stored cumulatively, avoiding the need of several copies (a homemade strategy used before the VCSs). A description of VCS can be found in [107].

There have been attempts by some researchers [29, 30] to measure the extent of the problem. They focused on verifying the availability of code and data first, and then if said code would build with reasonable effort. Out of the 613 papers they took into account, 515 were considered potentially reproducible, but the authors managed to run only 102 (circa 20%). It is worth noting that the authors simply attempted to run the code, without checking either its correctness, i.e., if a code implements what is claimed in the paper or if it produces correct output or the extent at which the results obtained in the original study were reproducible. An attempt to reproduce the results of a paper can be found in [129]. The authors tried to reproduce the results obtained using a model from computational neuroscience. Few pieces of information were available: no source code, neither as a complement to the paper nor in a publicly accessible repository. After contacting the authors of the original study, they could obtain the source code, only to discover that it could not be compiled and, in any case, it would be challenging to reuse it.

Recently, an initiative towards reproducible and sustainable Computational Science has been launched in the form of a peer-reviewed journal, ReScience [114], targeting computational research. ReScience encourages replication of already published research, this way enabling reimplementations to assess whether the original work can be replicated or not.

A typical practice, usually followed by scientists, is to write down the sequence of steps done or even implement some scripts to automate part of the tasks of executing experiments and then analyze the results. Several tools have been developed to help in exactly this task, focusing on the workflow of an application, i.e., interconnection of input/output and computation modules. As described by Maffia et al. [87], Taverna [95] and VisTrails [117] integrate data acquisition, derivation, analysis, and visualization as executable components throughout the scientific exploration process. The evolution of the software components used in the workflow is controlled by the organizations that design the tool. This approach is shared by other WFMS such as Kepler [86] and Knime [16], an open-source workflow-based integration platform for data analytics. All the tools listed are mostly oriented towards natural sciences but are not well suited for an HPC environment. There has been an effort to adapt

Taverna to the HPC context, through the OnlineHPC project [6], but the website is not reachable since middle 2016. Pathway [101] is a tool for designing and executing performance engineering workflows for HPC applications. DataMill [33] is a community-based easy-to-use services-oriented open benchmarking infrastructure for performance evaluation, which facilitates producing robust, reliable, and reproducible results. It provides a platform for investigating interactions and composition of hidden factors affecting the performance measurements, such as binary link order, process environment size, compiler generated randomized symbol names, or group scheduler assignments. Versioning systems such as git and GitHub are successfully applied in daily software engineering and document preparation. Sumatra [31] is a tool for managing and tracking projects based on numerical simulation and/or analysis, with the aim of supporting reproducible research: it can be thought of as an automated electronic lab notebook for computational projects. Even though versioning systems allow to share the code, its availability does not necessarily turn into reproducibility, due to the complexity of the software stack. PROVA! [57] overcomes this problem by taking care not only of the code, but also of the system, meant as a combination of hardware and software, and the metadata of an experiment (compilation flags, experiment parameters and results, that are fundamental contextual information for re-executing the experiment). Additionally, ACM has recently defined the 3 Rs of the Research [41] and their description overlaps with our three levels of reproducibility, first introduced in Guerrera et al. [55]. Moreover, we propose a standardized way of describing an experiment, which extrapolates a minimum amount of information needed for reproducibility purposes and a workflow to follow in order to carry out an experiment and achieve reproducible results properly.

1.3 Organization of This Work

This work has been structured into four parts. The first part includes an overview of the current computer architectures available, with all their peculiarities and limiting factors (Chapter 2), and the software characteristics and trends that have an impact on the reproducibility (Chapter 3).

The second part offers an overview of the proposed framework to address the challenges emerged in the first part. Having highlighted the factors that affect the reproducibility of the research in computational sci-

ences, a characterization of the experiments and an overview of how it can be coupled with reproducibility is offered (Chapter 4). Section 5.1 shows how to address the complexity, both at software and hardware level, for moving towards reproducible research. Follows an overview of the tool PROVA!, in Chapter 5.

The third part comprises the experimental evaluation of the concepts presented in the previous chapters of this work. First, the problem of stencil computation is introduced together with a tool to generate synthetic stencil benchmarks (Chapter 7). In Chapter 8 the concept of Performance modeling and evaluation are presented and after having characterized and described both hardware and software ((Chapter 9) used, according to the taxonomy introduced in Section 4.1, results of experiments in performance prediction are presented, with focus on the matching of said predictions and experimental performance results. Then, an in-depth analysis of a complete cycle of performance engineering is shown, following the approach proposed in Part II.

The fourth part, Chapter 11, draws the conclusions of this work, highlights some limit emerged and proposes ideas for follow-up studies.

Part I

Reproducibility Challenges

Chapter 2

Reproducibility Challenges: Hardware Complexity

While at the beginning of the supercomputing era the systems were custom made by producers like Cray, NEC, Fujitsu and were completely different by the standard computers both in terms of price and performance, since the beginning of the '90s single-chip general purpose microprocessors flooded the HPC market. At the time of writing, most of the microprocessors available in the supercomputers use off-the-shelf solutions, not explicitly designed for scientific computing, with few exceptions.

Turing has defined the paradigm at the basis of each modern microprocessor at the end of the '30s: the instructions, stored in memory, are read and decoded by a control unit, before being executed by a different unit responsible of the computation and manipulation of the data. Writing a program corresponds to save instructions into memory and make them available to the Central Processing Unit (CPU). Such an architectural model is still used in modern computers.

2.1 Moore's Law

Since the beginning of the digital computing era, the microprocessors have seen their capabilities (performance in a broader sense) doubling at regular intervals. Moore, observed in 1965 that the transistors count, on an integrated circuit, doubles every 18-24 months [96]. It is important to remember that Moore's law is a business statement, not a technology

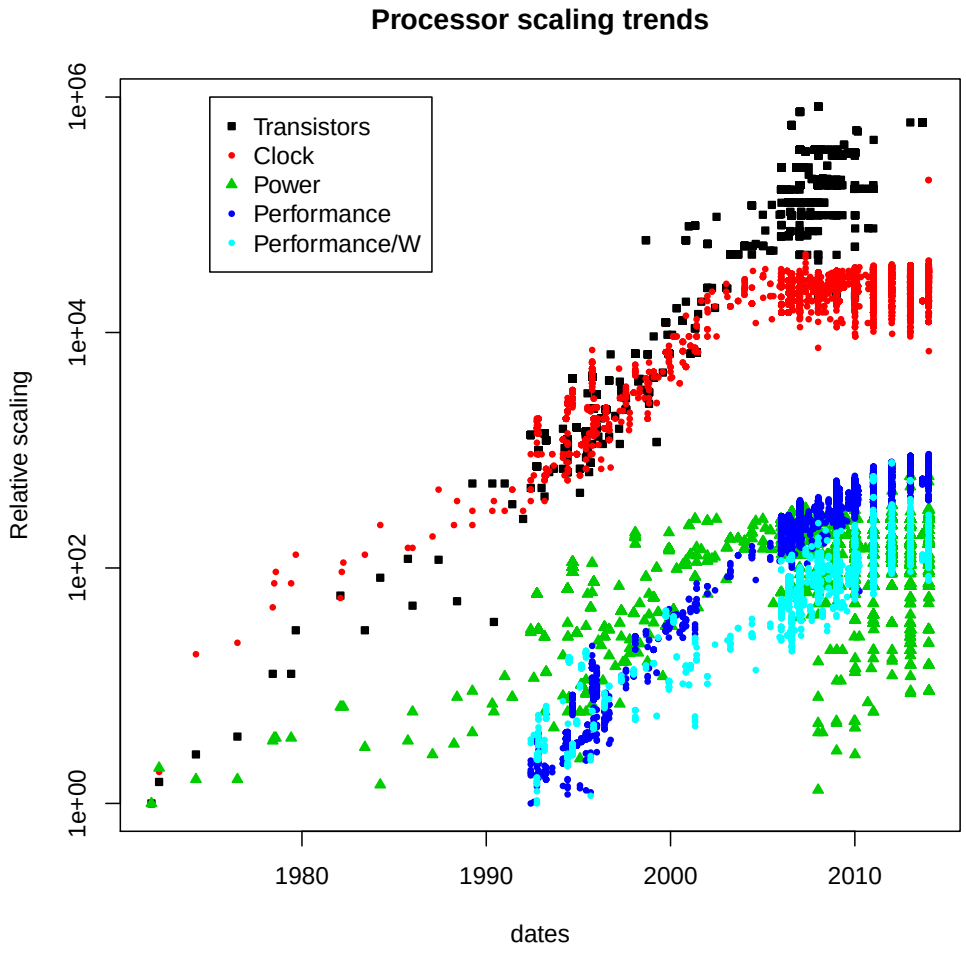

Figure 2.1: *Overview of the evolution of microprocessors: transistors count, clock frequency and power consumption. Note that the y-axis is presented in logarithmic scale and as relative value. Data taken from [1].*

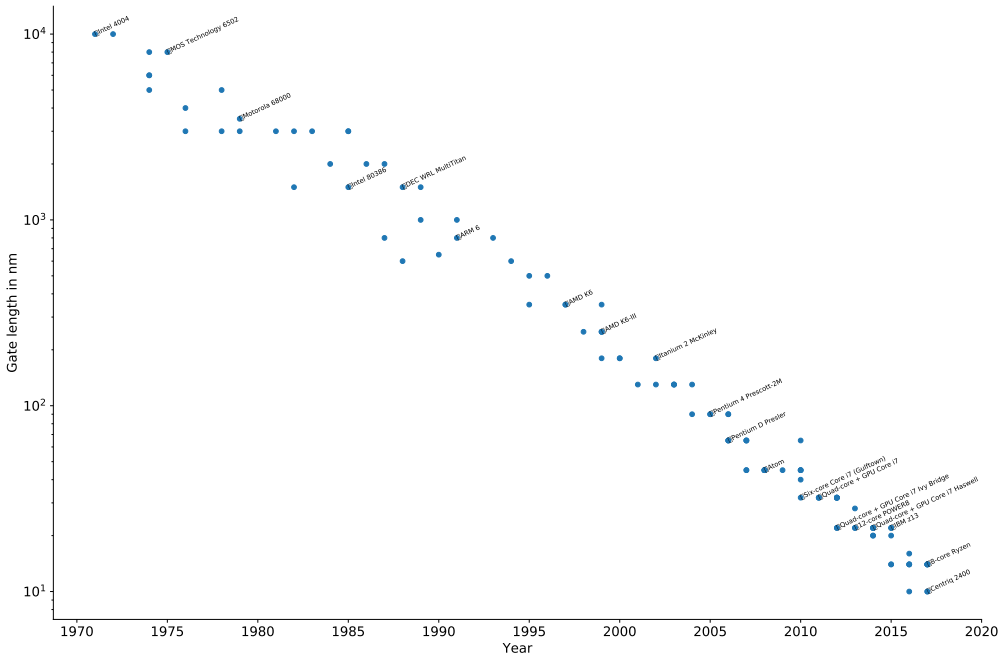

Figure 2.2: *Overview of the evolution of the gate length of the transistors used in microprocessors. Data taken from [4].*

law and it kept yielding due to an agreement between the manufacturers. As can be seen in Figure 2.1, circa until 2004 performance scaling came for free because of the advances in the semiconductor fabrication process that reduced the gate length of the transistors on the integrated circuit. In 1971, the Intel 4004 had a gate length of 10μm, while the technology at the time of writing allows having a gate length of 10 nm (e.g., Intel Ice Lake architecture). Figure 2.2 shows an overview of the evolution of the gate length in the transistors. The gate length reduction has notable side effects: its reduction of factor 2 requires the voltage to be half of its precedent value, the capacitance C also halves, thus the energy $E = CV^2$, reduces by a factor of 8. Thanks to the reduced gate length, the electrons have to travel shorter, so that the clock frequency can be doubled and the power consumption results to be:

$$P = f * E = 2 * f_{old} * E_{old}/8 = P_{old}/4.$$

Integrated circuits are built in 2D wafers, so that $P = P_{old}/4$ translates into the possibility to have four times more transistors, occupying the same area and the power consumption to be the same as before. Under these conditions, doubling the clock drives to double the performance, at

the same power consumption.

Moore's Law is often misinterpreted assuming that the compute performance doubles every 18-24 months. This is due to the performance being proportional to the clock frequency: thanks to the reduction of the gate length of the transistors, both clock and density of the transistors could be increased. Unfortunately, the consequent reduction of the voltage has drawbacks: the leakage power is increased. Additionally, semiconductors require a certain voltage threshold to work, so that there is a minimum voltage, under which it is not possible to downscale. In the above reasoning, if we keep the voltage constant and keep scaling the gate length, the power is increased by a factor 4, because the energy is proportional to the square of the voltage. As a consequence, manufacturers decided not to proceed further with the frequency scaling, since they reached the power that a consumer chip can handle (about 120 W). The green triangles in Figure 2.1 visualize the exponential increase in clock frequency until 2004, after which the curve flattens out. Current processor clock rates range around 2-3 GHz. The processor with the highest clock frequency ever sold commercially, IBM's zEC12 found in the zEnterprise EC12 System, runs at 5.5 GHz.

2004 represents the end of the frequency scaling era. At the time of writing, transistor gate length scaling in silicon-based semiconductors is coming to a halt as well due to the lattice constant (0.543 nm) of the silicon, making it almost impossible to scale further beyond the 10 nm technology node. Yet, Moore's Law is still alive thanks to technological advances. Semiconductor research is very active and exploring several new ideas: graphene-based transistors [84, 116], whose cut-off frequency around 3x higher than the cut-off frequency of silicon-based transistors; abandon the transistors in favor of novel components such as memristors [127]; or eventually moving away from using electrons towards using photons. Thanks to frequency scaling, processors have become increasingly faster, and the additional transistors have been used to implement out-of-order execution, Hyper-Threading, and branch prediction. Nowadays, the available transistors are used to create compute cores working in parallel. Indeed, the end of frequency scaling represents the beginning of the multicore era. Parallelism is no longer only hidden by the hardware, such as in instruction level parallelism, but is now exposing explicitly to the software interface. Current industry trends strive towards the manycore paradigm, i.e., towards integrating many relatively simple and small cores on one die. There are different ways of integrat-

ing the cores on a chip: they can share the caches or just some levels. As of the time of writing, architectures offer L1 private caches and start to share within a subset of cores the L2. The end of frequency scaling has also brought back co-processors or accelerators. Graphics processing units (GPUs) have become popular for general-purpose computing, and Intel has produced two generations of Many Integrated Core (MIC) architectures.

The memory remains the primary concern in moving towards an exascale system. While, as it has been described, the compute performance of the microprocessors used to double every 18-24 months, memory technology evolved but could not keep up. The consequence for it is the so-called memory gap: the latency between a main memory request and the request being served has grown into an order of hundreds of processor cycles. Additionally, the increase of the memory bandwidth has not been proportional to the compute performance. The consequence is that for a balanced computation, tens of operations on one datum are required. Many important scientific compute kernels (including stencil computations, sparse equation system solvers and algorithms on sparse graphs) have become severely bandwidth limited. In order to mitigate this issue, a hierarchy of caches, i.e., a hierarchy of successively smaller, but faster memories, has been designed, with the underlying assumption that data can be reused after bringing them close to the CPU.

Due to the memory gap, data movement is expensive in terms of energy, and more so the farther away the data has to be transferred from: data transfers have become more expensive than floating point operations. Therefore, data locality not only has to be taken seriously because of the impact on performance but also as an energy concern. Improvements in the near future include higher memory density and Hybrid Memory Cubes, i.e., stacked 3D memory cubes with yet higher bandwidth, lower power consumption, and higher memory density.

2.2 From Single Core to Multicore, Manycore, and Accelerators

The first compute system was built in 1949 upon the concept of the Turing machine. Instructions were stored in memory together with data. While the control unit reads and executes the instructions, the arithmetic-logic unit performs the computations and operates on the data and the instruc-

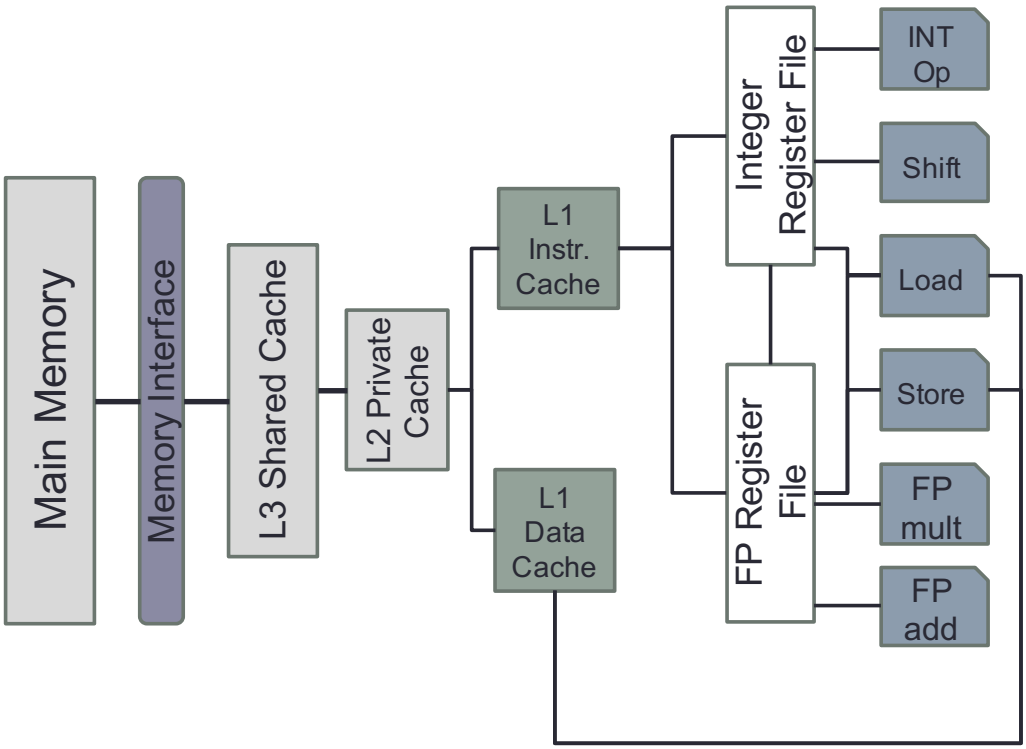

Figure 2.3: *Simplified schematization of a typical cache-based microprocessor.*

tions stored in the memory. These two units, together with the I/O interfaces, form the Central Processing Unit. Since the instructions (and the data) must move continuously from the memory to the CPU and back, the speed of the memory interface can limit the performance of the computation. A computer program is merely a sequence of instruction stored into the memory: humans usually write their programs in a higher-level language than the one understood by the machine. Compilers translate the programs into instructions that will be stored and executed by a computer.

Figure 2.3 presents a simplification of a real architecture. While the actual work of the execution of a program is done by floating-point (FP) and integer (INT) units, there is a whole set of additional units, whose purpose is to feed them with data. FP and INT units need data to be available in registers, so that they can operate. Load and store units make the data available to the registers and back to memory. To avoid moving the data back and forth, cache memory holds the data and the instructions that may be soon reused.

Once the frequency stopped scaling, the increased number of CMOS

started to be used for additional units like branch prediction, out-of-order execution, hyperthreading, as discussed in Sec. 2.1.

The disposable transistors have been used to develop advanced techniques and offer additional features like pipelining units, data parallelism (Single Instruction Multiple Data), out-of-order execution, superscalar capabilities, and caches.

2.2.1 Pipelining

Complex operations have been divided into smaller steps that can be executed by different units of the CPU. Each of the units executes the same operation as part of a chain. The concatenation of the single operation executed by each functional unit yields the final result. Ideally, every single task needs the same amount of time t to be executed: this way after each t a new result is available. At its easiest, the basic instruction cycle is broken up into a series of stages that form the pipeline:

1. Instruction fetch (IF): read the program counter (PC), take the current instruction and then update the PC
2. Instruction decode (ID): decode the instruction and read the register sources
3. Execution (EX): execute the instruction decoded (in case of load or store, compute the elective address)
4. Memory access (MEM): read or write to the memory according to the previous stage
5. Write-back (WB): write the results to the register file

As shown in Figures 2.4 and 2.5 the application of such a concept yields a five-fold speedup without changing the frequency of the clock. In both cases, the first result is available after five cycles (latency of the pipeline), but from that moment on, a result is available after each clock cycle (throughput of 1 cycle). Generalizing, executing N instructions results in $T_{seq}(N) = 5 * N$ cycles, in a sequential execution, while $T_{pipe}(N) = 4 + N$ cycles, in case of pipelined execution.

In such circumstances we obtain a speedup of:

$$\frac{T_{seq}}{T_{pipe}} = \frac{5 * N}{4 + N} \approx N,$$

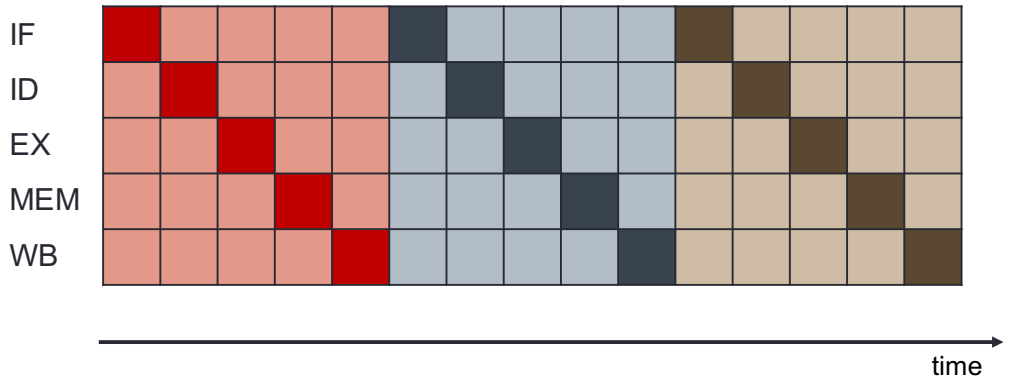

Figure 2.4: *Basic instruction cycle, broken into a series of stages, without pipelining. The light colors represent a single instruction, while the dark ones represent the active stage of the pipeline.*

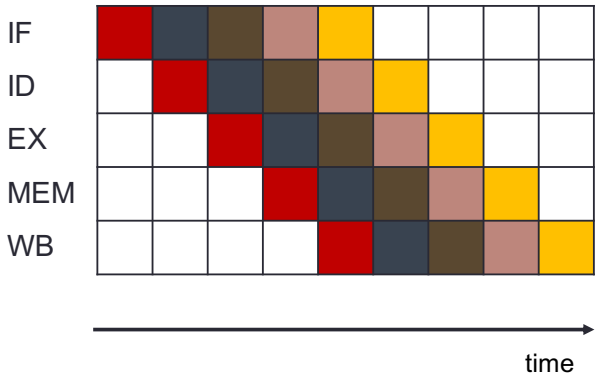

Figure 2.5: *Basic instruction cycle, broken into a series of stages, with pipelining. Each color represents a single instruction.*

for large N and throughput

$$Throughput = \frac{N}{T_{pipe}} = \frac{N}{N+4} \approx 1,$$

for large N.

It is worth noting that the pipelining concept, described for the instructions path, is also applied to the data path (actual computation). Besides the benefits, the introduction of the pipelining introduces some issues, related to the depth of the pipeline that must increase when the clock frequency increases. Additionally, the operations in the pipeline must be independent of each other: in fact, an operation can only take place if the operands are available. An example of the issue is described in [62]. Consider the following loop.

while $i < N$ **do**
    A(i) = s * A(i)
end while

Even if the operation can be pipelined, it is clear that the pipeline must stall if $A(i)$ is not available in time. One solution is to interleave different loop iterations (software pipelining), in order to avoid stalls and meet the latency requirements. Unfortunately, there are situations, called hazards, that prevent the pipelining of the instructions. Structural hazards arise from resource conflicts (i.e., a single memory unit needed for two or more instructions), control hazards occur from pipeline of branches and instructions that change the execution flow (addressed through branch prediction), and data hazards, that arise when an instruction depends on the previous result (Read After Write, Write After Read, Write After Write, and Read After Read).

2.2.2 Out-of-Order Execution

Pipelining provides higher performance by allowing the execution of different instructions to overlap. Performance can be further improved by allowing the instructions to be executed out of order. The out-of-order execution reorders the instructions exploiting register renaming, branch prediction, and multiple memory accesses. The instructions are fetched in a compiler-generated order and then dynamically scheduled.

2.2.3 SIMD

Historically, vector computers brought to the light the concept of Single Instruction Multiple Data (SIMD), and it has been revisited and reused over time. Recent micro-architectures offer in their instruction set an extension for both integer and floating-point SIMD operations. SIMD allows executing the same operation on a larger register, as shown in Figure 2.6, resulting in data level parallelism: parallel computations on a single instruction. It is worth noting that SIMD does not force the operation to be intrinsically parallel: if the arithmetic units are available, they can be (SIMD-) used for truly parallel computation or be fed with data from the pipeline.

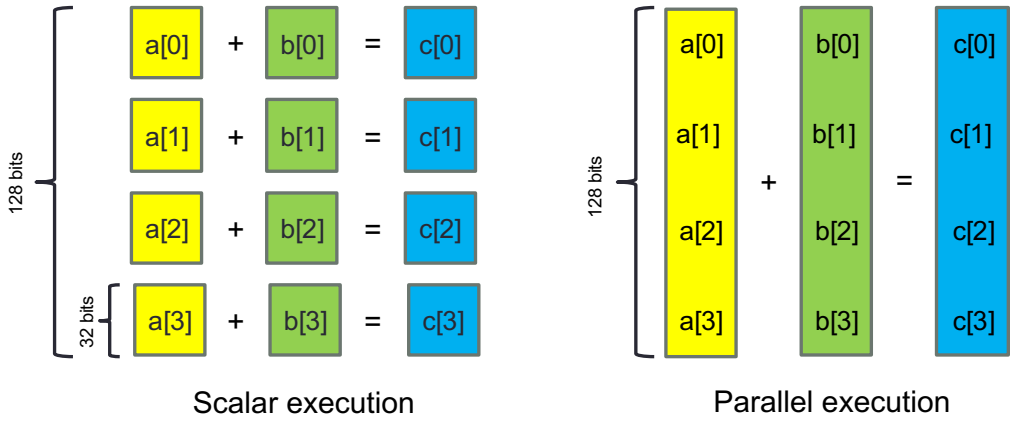

Figure 2.6: *SIMD operation compared to a scalar operation over an array.*

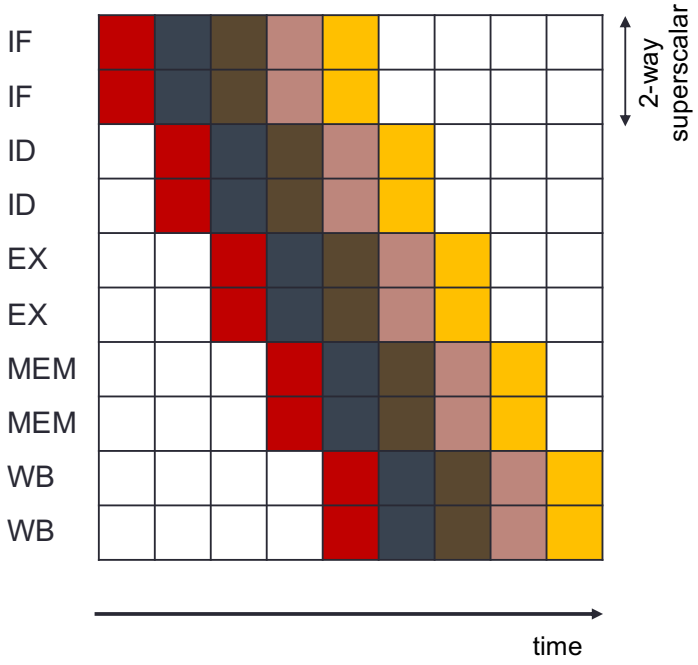

Figure 2.7: *Superscalarity of instructions: multiple functional units allow to complete execute several instructions per cycle. Each color represents a single instruction broken into stages in the pipeline.*

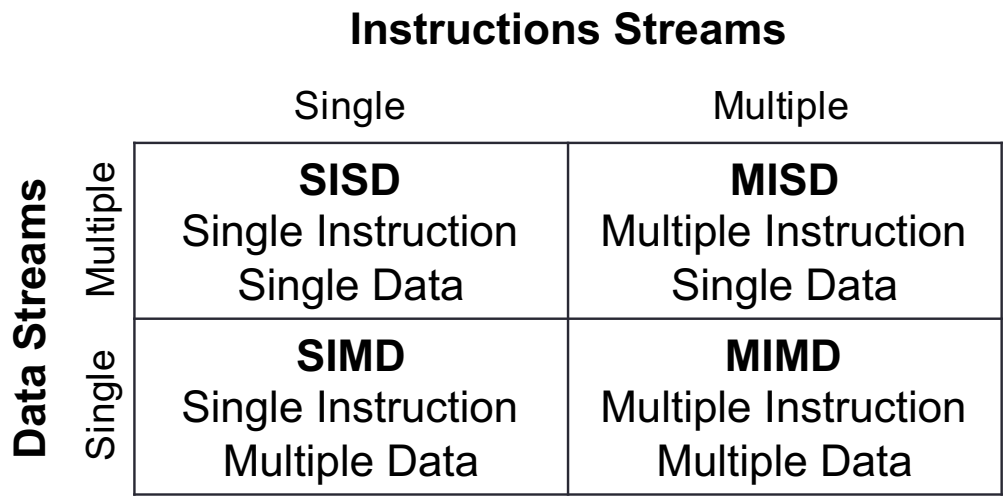

Figure 2.8: *Flynn's taxonomy of parallel computers, as described in [40].*

2.2.4 Superscalarity

The execution stage of the pipeline is a group of different functional units, each doing its task. As a consequence, it is possible to execute multiple instructions in parallel: several instructions can be fetched and decoded at the same time, multiple floating-point pipelines can run in parallel if the cache is fast enough to feed the required data. Multiple units enable the use of Instruction Level Parallelism (ILP): the instruction stream is "parallelized" on the fly. On superscalar RISC processors out-of-order (OoO) execution hardware is available to optimize the usage of the parallel execution. In Figure 2.7 there are 2 instructions completing every cycle, i.e. more throughput: CPI = 0.5. Modern processors are from 3- to 6-way superscalar and can perform 2 or 4 floating point operations per cycle.

2.2.5 Parallel Computers

In 1972 Michael J. Flynn proposed a classification of computer architectures based on instructions and data streams [40], as shown in Figure 2.8:

- SISD: single instruction, single data, schematizing the classical scalar uni-processor system, a sequential computer which exploits no parallelism in either the instruction or data streams
- SIMD: single instruction, multiple data. It can be achieved by pipelin-

ing or multiple functional units. It defines a vector architecture or vector processor.

- MISD: multiple instructions operate on single data. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result.

- MIMD: multiple instructions, multiple data. Different processors/-cores may execute different instructions on chunks of data. It represents the general multiprocessor.

In this work, when we use the words processor or multiprocessor, we refer to a MIMD multiprocessor.

2.2.6 Simultaneous Multi-Threading

At the time of writing, the majority of the processors is characterized by a high degree of pipelining: their performance is driven by the extent at which the pipelines can be used. As described above, in Section 2.2.1, the efficient usage of the pipelines is affected by memory latencies, branch mispredictions, dependencies, loop length, etc. These factors may drive the pipeline to be filled with bubbles, thus causing the idling of the resources.

The trend in computer design is to increase the length of the pipelines, thus allowing the clock speed to rise too. As a consequence, processors need more power without a considerable increase in average application performance. To overcome this issue, modern processors have threading capabilities, named Simultaneous Multithreading (SMT) or Hyper-Threading [90]. The main characteristic of the SMT design is the presence of several CPU cores, comprising status and control registers, stack and instruction pointers. Resources such as arithmetic units, caches, memory interfaces, etc. are instead not duplicated. The presence of several cores (in the CPU) allows the parallel execution of multiple instruction streams, often named threads, that can stem from the same or different programs. Since the threads share execution resources, it is possible to fill the bubbles in the pipeline combining instructions coming from different instruction streams. It is thus clear that SMT can enhance instruction throughput, whenever there is potential to interweave instructions from multiple threads within the pipelines.

SMT poses challenges related to the threads sharing resources like the caches, which could lead to an increase in capacity misses and synchronization issues [106]. Additionally, when SMT is present, affinity mechanisms (see Section 2.3.1) assume a crucial role.

2.2.7 Dynamic Frequency Scaling

Modern microprocessors can automatically adjust, "on the fly", their operational frequency, depending on the actual needs, to save power and reduce the heat generated by the chip. Frequencies can be scaled automatically depending on the system load, in response to ACPI events, or manually by userspace programs.

When performing reproducible benchmarking it is thus essential to keep the frequency fixed to a predefined value since the clock speed is one of the factors affecting the performance of an application. Frequency can be set both at OS level (see [2]), and by job schedulers (when the OS is configured to allow it).

2.3 Memory Subsystem

As previously mentioned, the CPU acts on the data available in the registers. Between the registers (fast, expensive and extremely small) and the main memory (slow, cheap and large) there are variable levels of cache memories (small but very fast), integrated on the die, that store copies of the recently used data. Data are used by the CPU, but must first be moved to the registers from the main memory. Memory bandwidth is defined as the rate in bytes per second at which the data is transferred from memory to the CPU. Memory latency is, instead, related to the response time, i.e., how many clock cycles the memory will delay in returning data requested by the CPU.

While current machines have a pick performance of various Gflop/s per core, the memory bandwidth is few GBytes/s, meaning that it is not able to feed the CPU with enough data to avoid it being idle. Additionally, there is also the memory latency to worsen the situation. The term memory gap refers to the increasing divergence of the performance between CPU and memory sub-system [65, 88].

As shown in Fig 2.3, at the time of writing, computer architectures consist of 3 levels of cache. The first level, called L1, is private (per-core)

and usually divided into a part designated to the data and one to the instructions. The second level, called L2, is either private per core or shared with a small subset of cores. L3, the level closest to the main memory, is usually shared per socket. From L1 to L3 the dimension and the latency increase while the bandwidth decreases. When moving data from the memory to the registers, it must pass through each level of cache, where a copy of it will be hosted. Later, if the CPU needs a certain data that is hosted in one of the cache levels, a cache hit occurs, otherwise (data not available in cache) a cache miss takes place. When the cache is full and the CPU requests new data, such data will replace some other one into the cache (according to a replacing policy), thus having an eviction.

Many applications exhibit a pattern of access to the data that shows a locality of reference. Data loaded into the cache are most likely to be reused soon, before their evict has been issued. Such a phenomenon is called temporal locality and, as described in [62], if the ratio of data reuse is close to one, a cache system leads to significative performance advantage. Instead, other applications show a different access pattern requesting large amounts of data to be loaded, modified and written back without potential reuse in time. Such a pattern is called streaming. In this case, the reuse ratio would be close to 0, thus showing no benefit in using a cache system.

Caches are organized in lines: all data transfers are based on the unit of cache lines, thus mitigating the latency for streaming applications. Since a line is loaded as a whole from memory, items that are stored close to each other, are loaded with a much lower penalty in terms of latency. If an application accesses items that are close to each other, thus showing spatial locality, the latency penalty can be reduced significantly. Unfortunately, the cache system does not bring benefits to applications that display a random access pattern.

2.3.1 Shared Memory Computers

A parallel computer is defined to have shared memory when a certain amount of CPUs share a physical address space. Two main categories of shared-memory architectures are available: UMA, shown in Fig 2.9, and ccNUMA, shown in Fig 2.10. In the Uniform Memory Access (UMA) systems, memory latency and bandwidth are the same for each of the processors. Such an architecture is scalable only for a limited number of processors.

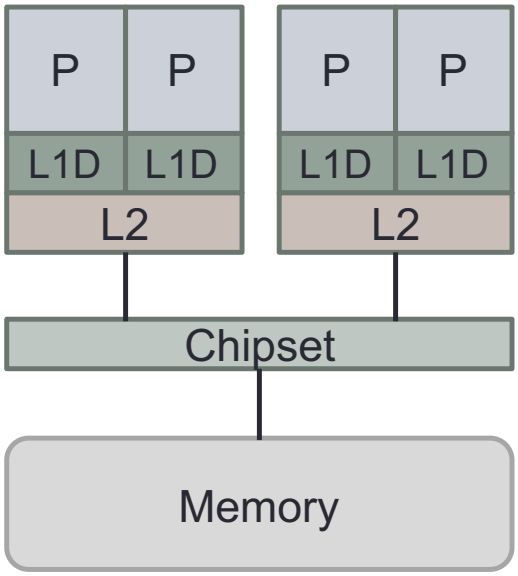

Figure 2.9: *UMA system consisting of two dual-core chips.*

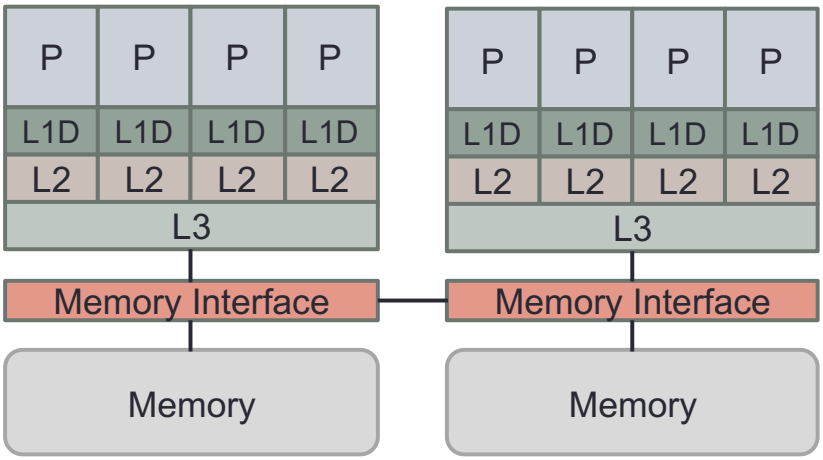

Figure 2.10: *ccNUMA system consisting of two NUMA domains and eight cores.*

In the cache-coherent Nonuniform Memory Access (ccNUMA) systems the memory is logically shared but physically distributed. Each processor has its own local memory, and the memory of other processors is accessible, even though is more far away, thus resulting in different latencies and bandwidth depending on the distance of the memory from a processor. A locality domain consists of a set of cores and their locally connected memory. Multiple locality domains are connected through a coherent interconnect, that allows the cores in one domain to access the memory of the cores in another domain. It is essential that the network connection has bandwidth and latency at least as good as the local ones. The penalty for accessing a non-local domain can severely affect the performance of a code. That is particularly true for memory bandwidth bound application: locality and contention problems show up when the data is not correctly placed across the NUMA domains. One of such issues is due to the page placement: the data should be mapped into the local domain of the processors using them, in order to minimize the traffic on the links. Additionally, the threads must be pinned to the cores (CPUs) where the data has been initially mapped, in order to avoid losing locality (due to a thread that migrates from one CPU to another). ccNUMA architectures support a first touch policy: the memory pages are mapped into the locality domain of the processor that first writes (not merely allocates) to it. Consequently, one must pay attention to the data initialization on ccNUMA systems. The pseudocodes in Listings 2.1 and A.6 show the wrong and correct way of initializing an array while calculating a three-dimensional isotropic star stencil with radius 1, respectively.

```
1:  double *a = malloc((sizeof(double))*((M*N)* P));
2:  for (int i = 0; i < ((M * N) * P); ++i)
3:    a[i] = rand() / ((double ) RAND_MAX);
4:
5:  double *b = malloc((sizeof(double)) * ((M * N) * P));
6:  for (int i = 0; i < ((M * N) * P); ++i)
7:    b[i] = rand() / ((double ) RAND_MAX);
8:
9:  ...
10:
11: #pragma omp parallel for schedule(runtime)
12: for (int k = 1; k < (M - 1); k++)
13: {
14:   for (int j = 1; j < (N - 1); j++)
15:   {
16:     for (int i = 1; i < (P - 1); i++)
```

```
17:       {
18:         //pseudo
19:         b[k][j][i] = c0 * a[k][j][i] + c1 * (
20:           (a[k][j][i-1] + a[k][j][i+1]) +
21:           (a[k-1][j][i] + a[k+1][j][i]) +
22:           (a[k][j-1][i] + a[k][j+1][i]));
23:       }
24:     }
25:   }
```

Listing 2.1: *NUMA unaware initialization of the arrays used to calculate a a three-dimensional isotropic star stencil with radius 1*

```
1:  double *a = malloc((sizeof(double))*((M*N)* P));
2:  #pragma omp parallel for schedule(runtime)
3:  for (int i = 0; i < ((M * N) * P); ++i)
4:    a[i] = rand() / ((double ) RAND_MAX);
5:
6:  double *b = malloc((sizeof(double)) * ((M * N) * P));
7:  #pragma omp parallel for schedule(runtime)
8:  for (int i = 0; i < ((M * N) * P); ++i)
9:    b[i] = rand() / ((double ) RAND_MAX);
```

Listing 2.2: *NUMA aware initialization of the arrays used to calculate a a three-dimensional isotropic star stencil with radius 1*

Figure 2.11 shows the performance of the code with faulty initialization, on the left, and the one of the code correctly initialized (right). It is clear the impact of such a simple modification. One of the possible problems with such initialization is related to the OpenMP loop scheduler: initialization and work distribution must be identical, i.e., a STATIC loop schedule is needed. This is the default choice of OpenMP. Another issue could raise if the hardware is not capable of scaling the memory bandwidth across the locality domains.

The problem of the thread pinning has been showed in [59], where several experiments have been conducted with and without thread pinning, using different pinning strategies. As a mean of pinning has been used the Likwid tool [112, 130]. The results published show that without the correct pinning, the performance suffers both in terms of absolute value, and in variability across several runs. The issue of thread pinning arises since some operating systems leave the threads free of moving from one CPU to another (level of affinity between thread and processor). Erratic performances are an indicator of insufficient thread affinity.

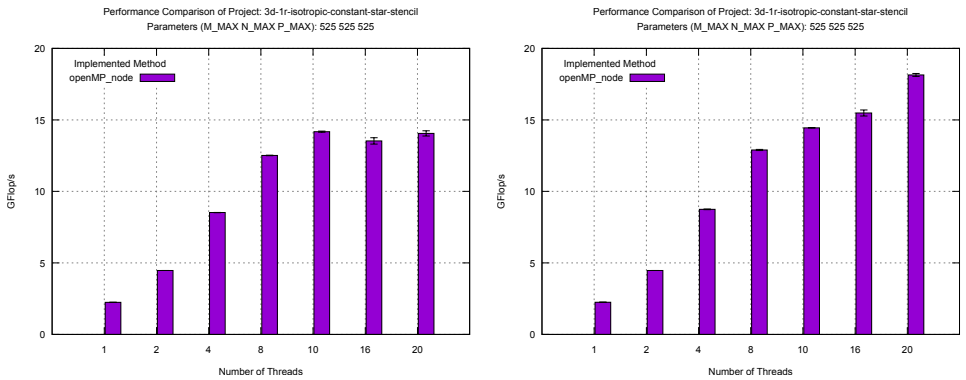

Figure 2.11: *Performance of the wrongly (left) and correctly (right) initialized three dimensional isotropic constant stencil with radius 1.*

2.3.2 Real Life Intermezzo

During the time spent writing this dissertation, I myself faced a reproducibility issue, while running a 2-dimensional isotropic box stencil with constant coefficients (described in Section 9.3) and radius 1, with input size of the grid 10000^2, NUMA unaware initialization, and thread pinning to the logical nodes via likwid. The code was run on a node with the following hardware characteristics:

- 2-socket Haswell chips
- Intel Xeon E5-2695 v3 – 2.30 GHz (fixed for all measurements) – AVX2 (FMA)
- 14 cores per chip; 32 kB L1; 256 kB; (17 + 17) MB L3
- Cluster on Die: Enabled (4 NUMA domains with 7 cores (17 MB L3) each)
- Dynamic Frequency Scaling: Disabled
- 8 * 8 GB DDR4-2133 / 4 channels
- Peak BW= 68.3 GB/s
- Single core peak performance: $2.3 * 2 * 2 * 4 GFLOP/s = 36.8 GFLOP/s$
- Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-57-generic x86_64)
- Intel Compiler: icc 15.0.1 20141023
- Thread pinning via: likwip-pin (to logical nodes)
- CPU frequency fixed to baseline frequency (likwid-setfrequencies)

In Figure 2.12 are shown the results obtained on the same machine, with a temporal distance of a week (on the left the most dated results). As the reader can imagine, said results surprised the author, unable to find a reasonable explanation to the differences. Interestingly enough, the

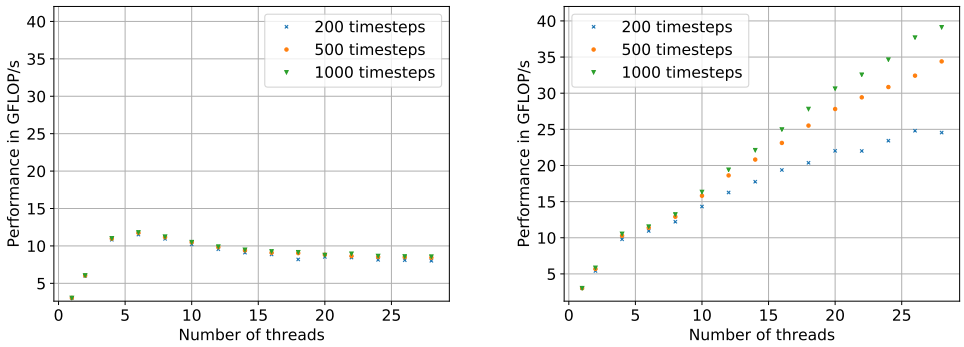

Figure 2.12: *Performance of a 2-dimensional isotropic box stencil with constant coefficients and radius 1, grid size of* 10000^2 *and NUMA unaware initialization. On the left the performance, varying the numbers iterations, on an Intel Xeon E5-2695 with Cluster on Die mode enabled and NUMA balancing disabled. On the right the performance of the same code, on the same machine, but with NUMA balancing enabled.*

results on the right (most recent execution) show how the performance of said stencil varies while increasing the number of iterations. In Figure 2.12, on the left, the performance stops scaling at six threads (cores), due to the saturation of the socket. PROVA!, as described in Chapter 5, stores contextual information of the experiment: not only software used but also details of the machine, where the code executes. Having a look at these files, a difference in a single parameter popped up, i.e., the value of the NUMA balancing. An application will generally perform best when the threads of its processes are accessing memory on the same NUMA node as the threads are scheduled. Automatic NUMA balancing moves tasks (which can be threads or processes) closer to the memory they are accessing. It also moves the application data to the memory closer to the tasks that reference it. After noticing the change in the value and with further investigations, turned out that the machine had been restarted and the value set to its default value of 1.

2.4 Network Subsystem

Together with hardware and software, the third fundamental component of high performance computing is represented by the network interconnect. A large variety of solutions is available, with the cheapest one being the Gigabit Ethernet. The trend is represented by optical interconnects

that tend to provide a wider bandwidth while being more efficient.

At the time of writing, the #1 machine in the TOP500 [9] listing, named Summit, is composed of 4356 nodes: it is self-evident that central switches cannot be used anymore, and hierarchical structure should be preferred. Summit utilizes a dual-rail Mellanox EDR InfiniBand interconnect, with a non-blocking fat-tree topology, for both storage and inter-process communications traffic. It interconnects thousands of compute nodes containing both IBM POWER CPUs and NVIDIA GPUs, delivering 200Gb/s network speed to each of the compute platform. The advances in the InfiniBand technology allow the applications to communicate latency-sensitive data effectively.

MPI [42] is one of the programming models that allow the applications to communicate over a network interconnect, by exchanging messages. Part of the network subsystem is used to carry out the message exchange, thus demanding a good mapping between the hardware and the communication requirements of the applications. Such a problem is becoming more and more relevant since, as of now, the supercomputers scale by increasing the number of nodes and their heterogeneity: it inevitably affects how the systems are programmed and must be addressed on the software side.

Chapter 3

Reproducibility Challenges: Software Complexity

In the previous chapter has been described how the hardware architectures have evolved, showing the complexity of the machines that are available at the time of writing. The manycore paradigm is the rule, and the application developers must exploit the large amount of parallelism offered, thus directly affecting the software. Both inter-node and the fine-grain intra-node parallelism must be dealt with and addressed. On the programming side, one must consider data locality as well as synchronization, trying to reduce communication.

3.1 Amdahl's and Gustafson's Laws

Since computer manufacturers are providing architectures with an increasing number of computing cores, applications must be parallelized to properly exploit all the available processors. Ideally, when passing from running a program on a single compute core to running it on N cores, one may think of obtaining a reduction of the execution time t to t/N. Let us denote with f_{seq} and f_{par} the sequential and the parallel part of a program, respectively. If the parallel part can be made N times faster by using N processors, then the time to solution is:

$$T_N = f_{seq} * T_1 + \frac{f_{par} * T_1}{N}.$$

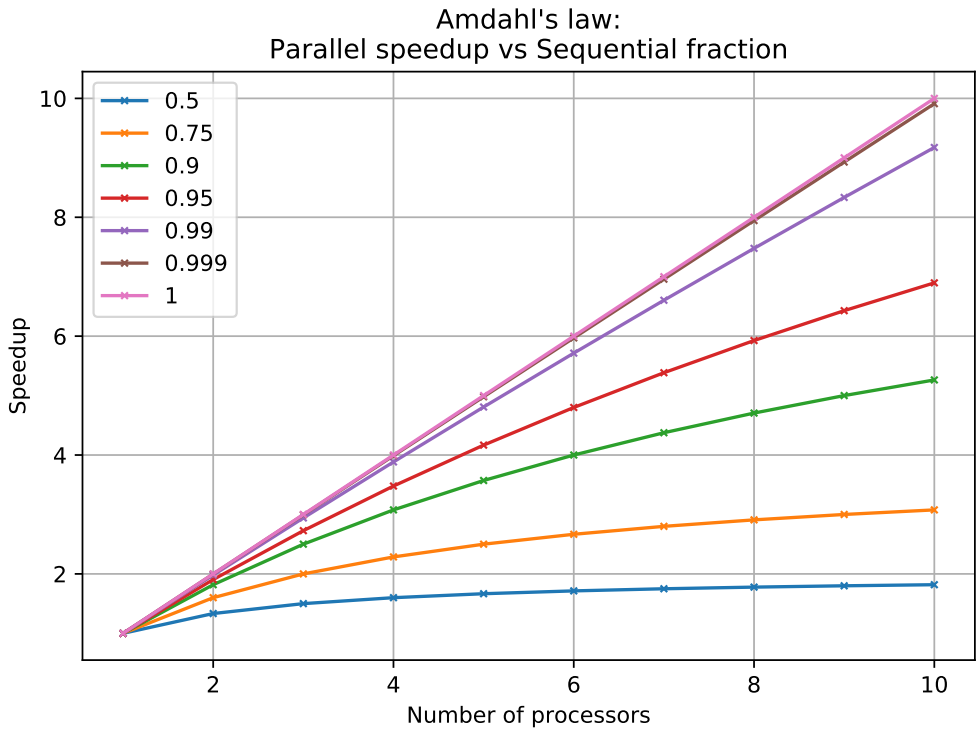

Figure 3.1: *Amdhal's law [10]: parallel speedup vs sequential fraction, for ranges of the parallel fraction between 0.5 and 1.*

Amdahl's law [10], states that given a fixed problem, the speedup (i.e. the ration between the original and the new execution time) of a parallel machine with N processors is:

$$S_{strong}(N, f_{par}) = \frac{1}{\frac{f_{par}}{N} + (1 - f_{par})},$$

where $f_{par} = 1 - f_{seq}$ represents the fraction of the program that can be parallelized. In Figure 3.1 is shown how the speedup would look like when f_{par} varies between 0.5 (half of the program can be parallelized) and 1 (the whole program can be parallelized). The consequences of Amdahl's law are dramatic: inamely, a non parallelizable fraction of the original code $f_{seq} = 0.25$ limits the speedup to:

$$S_{strong}(N, f_{par}) = \frac{1}{0.25} = 4.$$

Such a forecast represents the reason why Amdahl's law is usually defined as pessimistic.

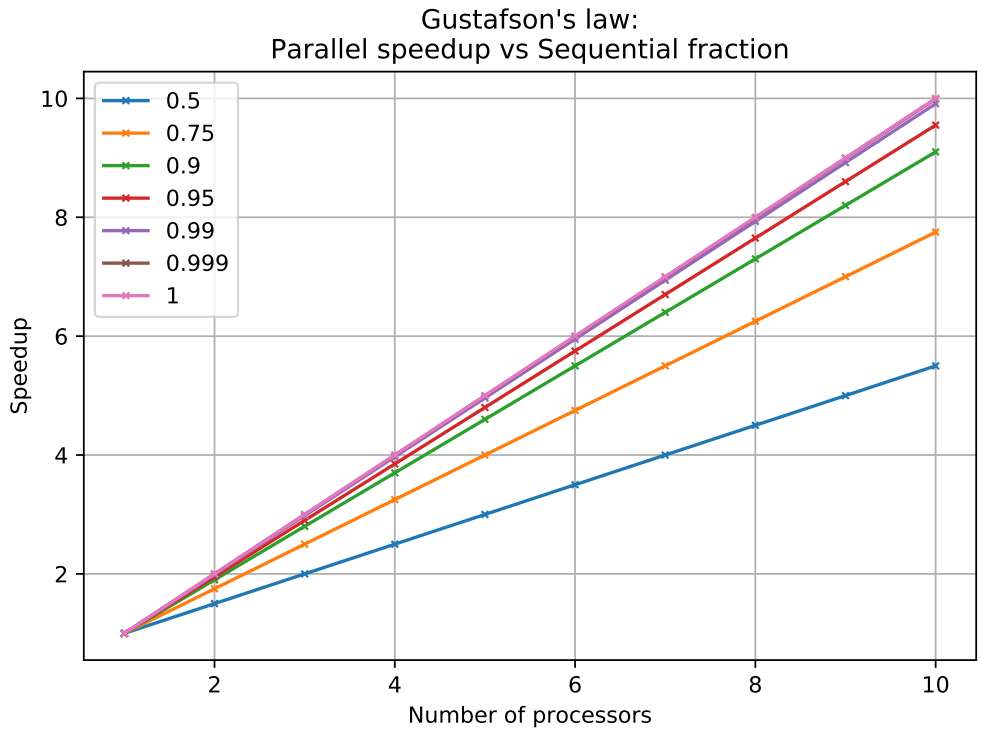

Figure 3.2: *Speedup as a function of the number of cores for the ranges of the parallel fraction 0.5 to 1 assuming Gustafson's law [60].*

In 1978 Gustafson approached the problem from a different perspective: rather than fixing the problem size, as for Amdahl's assumption, one may fix the time to solution. In such a condition, the result is scaling up the size of a problem, assuming that the size that can be solved grows together with the available parallelism. Such a way to approach the scaling is defined *weak scaling*, opposed to the one where the size is fixed, defined *strong scaling*. In Gustafson's scenario, the speedup is calculated as:

$$S_{weak}(N, f_{par}) = N * f_{par} + (1 - f_{par})$$

which represents a much more optimistic view than Amdahl's, as can be seen in Figure 3.2.

3.2 Programming Models

The majority of the HPC applications are written in C, C++ or Fortran. None of these languages directly offers constructs for parallel execution*, which is offered instead via external libraries (e.g., POSIX threads).

At the time of writing, OpenMP [99] is the preferred option to exploit thread-level parallelism: its API has become the de-facto standard for shared-memory computers and intra-node concurrency. It utilizes compiler directives, thus resulting independent from the programming language. Such a choice though makes it dependent on the compiler supporting it. Nonetheless, thanks to its directives (pragmas), an OpenMP program can still be correctly compiled to a sequential program, if the compiler does not support OpenMP. It offers support for loop level and task level parallelism. It provides a higher level abstraction compared to pthreads thus making it easier for the programmer to implement an application without having to manually take care of the synchronization (as it would have been while using semaphores). OpenMP is restricted to shared memory environments.

OpenMP offers support for accelerators only from its version 4.5 [99]. In the meanwhile, other languages directly targeting accelerators have been proposed and accepted by the community, such as OpenACC (a directive-based programming model targeting a CPU+accelerator system, similar to OpenMP), CUDA (an explicit programming model for GPU accelerators) and OpenCL. In programming models, research has been focusing on an efficient use of intra-node parallelism, able to properly exploit the underlying communication system through a fine grain task-based approach, ranging from libraries (Intel TBB [109]) to language extensions (Intel Cilk Plus [110] or OpenMP), to experimental programming languages with focus on productivity (Chapel [25]). Kokkos [23] offers a programming model, in C++, to write portable applications for complex manycore architectures, aiming for performance portability.

HPX [77] is a task-based asynchronous programming model that offers a solution for homogeneous execution of remote and local operations.

Message Passing Interface (MPI) [42] is the de-facto standard for programming distributed memory architectures. Its first release is dated 1994, and now the standard has reached its version 3. MPI offers sev-

*C++ 17 defines execution policies and implements a parallelized version of some algorithms, but needs support from the compilers (missing at the time of writing)

eral communication styles and, thanks to its execution paradigm, the programmer knows how the data are distributed and where the code is executed. Communication and synchronization are performed through calls to the MPI library. The communication can be one- or two-sided (i.e., one process issues a send to the receiver process, which must call the *recv* function, otherwise the program is in a deadlock). The overhead for the programmer is quite substantial: having a fine-grain control over the program and its execution, means that the programmer must control all the aspects of the application code, making this programming style very powerful but error prone.

3.3 Compilers

Compilers are needed to translate a high-level language into a code that is executable by a computer. Optimizing compilers often bring benefits, in terms of performance, to the code they are applied to, being the reduction of time to solution (i.e., performance) one of the goals of the optimization. The role of an optimizing compiler comprehends the decision of which part of the code to optimize, check that the optimization is allowed (i.e., does not change the semantics of the original code) and eventually, translate that portion of the code. Mapping the source code to the machine code is not an easy task, and the compiler must carry it out carefully and properly, utilizing the resources and characteristics (highlighted in Chapter 2) of the processor it is compiling for. In fact, an optimization often represents a trade-off: optimizing for ILP or for reducing computation time, could result in a higher pressure on the registers or increased risk of capacity misses. Keeping track of the optimizations applied to code is fundamental when aiming to reproducibility, due to the impact that said optimizations could have on the final performance of the code.

3.3.1 Basic Optimizations

Compilers generally offer a set of basic optimizations, divided into levels. The kind of optimizations applied for each level is non-standard and varies from compiler to compiler. Usually, at level 0 (*-O0*) no optimization is applied, thus producing the easiest possible machine code. Higher levels of optimization, *-O1* to *-O3*, act on memory access transformations, such as register allocation, removing cache set conflicts or false sharing of cache lines.

The technique called inlining consists in the substitution of a function call with code directly injected at the place where it is called. The result is that the arguments of the function do not need to be pushed to the stack, but can be directly stored into the registers, with a possible drawback of higher register pressure. Abusing of the inlining can result in bigger object code, that can cause issues with the L1 instruction cache capacity.

The aliasing, namely the possibility to access the same memory location using different names, is one situation, driven by the programming language adopted, that limits the compilers' ability to generate optimized code. While C and C++ allow pointers aliasing, Fortran forbids it, thus resulting in faster code than the equivalent C/C++ version. For this reason, C/C++ compilers offer the option to control the aliasing with a command-line switch (*fargument-noalias* in GCC, *fno-fnalias* when using the Intel compiler).

Floating point expressions are usually not rearranged, except when using aggressive optimizations, because of their non-associativity.

It is a task of the compilers to orchestrate the registers, putting into them the operands that are used the most, and keeping them there as long as possible. Inlining, as described earlier, helps the registers optimization because the arguments do not need to pass through the stack, thus forcing a read/write to memory. When a compiler deals with loop bodies abundant in variables and arithmetic expressions, it can quickly run out of available registers to hold all the operands: in such a case, the variables must be written back to memory, thus resulting in a performance loss.

The impact of different compilers flags can be appreciated in Figure 3.3: the code executed is the same, with the only difference being a different set of compilation flags, resulting in two different behaviors at runtime. Such an experiment shows the importance of details that should be shared together with the performance results of a code and motivates a precise description of the experiments, as discussed in Chapter 4.

3.3.2 Loop Optimizations

Most of the work, in scientific codes, is carried out in loops, thus loop transformations are operations that can result in significant performance gains. Generally, loop transformations aim to: increase data locality, parallelization, exposing parallelism, decrease register pressure, increase in-

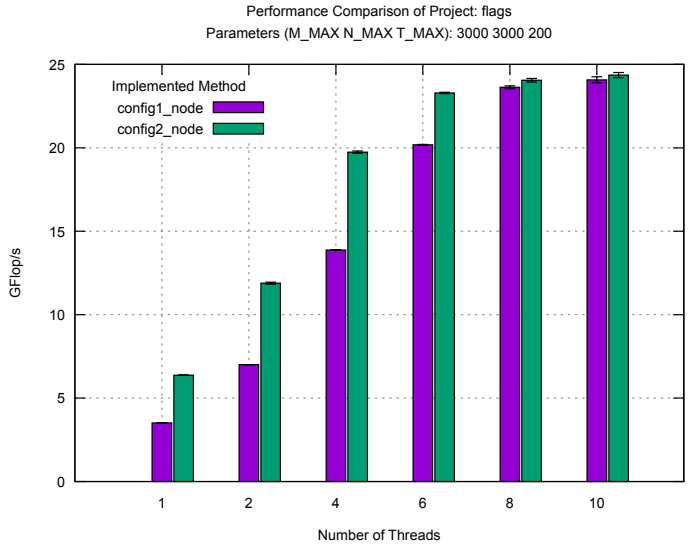

Figure 3.3: *Runtime performance of a 2-dimensional isotropic box stencil with constant coefficients, on a node of miniHPC (see Section 9.1.4), using explicit thread pinning. The input grid size is fixed to* 3000^2*, and 200 time-steps are executed in double precision. The difference in the performance of the two configurations is given by a different set of compilation flags used.*

struction level parallelism, decrease loop control and computation overheads.

An increment of data locality can be obtained through loop tiling, i.e., splitting the iteration space of a loop in smaller tiles. The ideal choice for a tile is the one that allows the data to fit into the cache, thus avoiding expensive cache misses. Another way of increasing the locality is provided by rearranging the order of the loops (*loop interchange*), letting the innermost one iterate over the unit stride dimension, thus maximizing cache line reuse.

Loop parallelization can be achieved exploiting data parallelism (i.e., vectorization) or task parallelism. Sometimes it is needed to apply a loop transformation first, in order to expose the parallelism of the original loop nest. It can be achieved via loop interchange or loop skewing (i.e., reshaping the iteration space to avoid dependencies).

To reduce the register pressure, a loop can be split into two or more loops, thus preventing the spilling of the variables into the memory. Nevertheless, such a choice can causexs an increase in the loop control overhead.

Loop unrolling represents a way to increase *ILP* and, at the same time,

decrease the loop control overhead. Its drawback is the increased register pressure. Decreased loop control overhead and increased register pressure is the result of another optimization too, *loop fusion*, that combines two or more loops into one.

Moving some loop invariant expressions, i.e., independent from the loop index variables, outside the loop to avoid that they are computed several times, is an optimization that decreases the loop computation overhead, at the cost of increased register pressure.

3.4 Software Stack

While for standard desktop systems, it is common and often enough to have a single version of a software package installed, this does not yield for HPC systems, which usually have a vast group of users with different needs. Furthermore, multiple version of the same software may include competing packages providing some functionalities. In [51] it is shown how the output of a neuroimaging pipeline can change when the OS version differs on the same architecture, due to a different version of the *glib* library available on them, and how it varies when using different architectures with the same OS. Changing the version of a single software used to conduct an experiment is enough to enter the field of irreproducibility, like reported in [53]. Reproducibility issues arise when variations of hardware and software (including its compilation) used to execute a code are not carefully noted, thus impeding to later identify the factors that can affect its performance.

3.4.1 Environment

The environment modules [45, 46] provide a solution to the issue of having several versions of a specific software package all available on a system. They allow a user to customize his environment, by loading or unloading the required software packages. Software modules work manipulating environment variables such as *$PATH* and *$LD_LIBRARY_PATH*, pointing to the locations of binaries, header files or libraries. One approach could be to provide a script for each piece of software, and the users must source it before using a specific package. Environment modules act exactly like that, but transparently to the users, allowing them to construct their environment dynamically by loading applications, libraries, and compilers. The original implementation [45], was a simple

collection of shell scripts, while today modules managers can be written in C with a Tcl interface, or Tcl-only, or C-only, offering more advanced funcitonalities. A newer approach to software modules is represented by Lmod [93, 39], implemented in Lua [73] and compatible with Tcl-based modules. Its goal is to allow users to have real control over their working environment while improving the user experience without affecting the experts. Lmod tries to overcome some of the drawbacks of the existing module tools, like the issue arising when loading modules with conflicting dependencies, thanks to the concept of families. It is possible to assign properties to the modules, an essential feature to tag modules that can run on accelerators.

As clarified in [48], programs compiled with a certain compiler version may experience errors when linking to libraries compiled with another. Furthermore, there are some software packages like MPI that are coupled with a particular compiler or even compiler version.

3.4.2 Build Process

Software modules represent, at the time of writing, the standard solution adopted in HPC centers. The software modules approach comes at the cost of a huge complexity in managing them when their number increases, as well as installing the software packages they refer to. The installation is a non-trivial task on its own: scientific software is often written by domain experts, whose focus is not to provide a clear and robust build procedure across several architectures. When it comes to installing such software packages, the solution can be: a manual installation by a highly trained and specialized user support team, a collection of script that automates the tedious sequence of configure-build-make (hard to maintain long-term solution), package managers, e.g. RPMs and yum for RedHat based systems, manual module files creation, custom-tools.

While the usage of package managers simplifies and makes it easy to configure and install specific software correctly, it is not a suitable solution to manage multiple version of the same one, whereas manually creating module files is not a long-term solution due to the module files easily losing consistency.

Custom tools are often proposed, starting from a collection of locally used scripts. Few survive, when attracting the interest of the community and in case they are flexible enough and well documented. EasyBuild is one of such projects, and “aims to provide an easy yet powerful way

to automatically install (scientific) software stacks, in a robust, consistent and reproducible way" [48]. It is a collection of Python modules that offer functionalities to install scientific software on HPC systems, including loading of module files and dependencies, managing the build environment and creating the module files for the software installed through it. It provides the concept of *compiler toolchain*, consisting in a set of compiler and libraries often used together. Each software package is meticulously described through a so-called *easyconfig*, i.e. the specification of build parameters and actions such as *name*, *version*, *dependencies*. Thanks to its characteristics it has rapidly attracted interest in the HPC community and is used across several HPC facilities in Europe, including the Swiss Supercomputing Centre (CSCS) hosting Piz Daint, the 6th most performing supercomputer in the world [9] (at the time of writing). EasyBuild ensures reproducibility and preserves provenance because each aspect of the installation, from the configuration to the build and the environment at each time is under control: a log of the whole process is stored together with the resultant module file, the sources of the software and a file containing the specification of the package and its dependencies. Spack [47] represents an alternative to EasyBuild: a flexible, configurable, Python-based HPC package manager, automating the installation and fine-tuning of simulations and libraries. Unlike EasyBuild, Spack uses RPATH linking to ensure that each package can retrieve its dependencies, and generating a module configuration file. EasyBuild and Lmod teams have built great synergy: thanks to the first installing scientific software has been eased resulting in need of managing the modules environment, through a modern module tool.

Part II

The PROVA! Approach to Reproducible HPC Research

Chapter 4

Proposed Framework

In the previous chapters have been highlighted the hardware and software characteristics of the architectures currently in use in HPC systems. Chapter 2 presented a brief history of the evolution of the computers from their invention until the present, trying to explore their capabilities and the actual strengths and limits of their architectural designs. The abundance of cores and functional units enforces the users to think of the software in an entirely new way, often having to re-write the algorithms to expose parallelism. The software packages have become more and more complex, all over their life cycle, from the installation to their usage. The resulting variegated picture poses severe challenges to the reproducibility of the research carried out using such a complex mixture of hardware and software.

In Table 4.1, is presented a short list of the most important publications and tool related to the topic of reproducibility. By using the words of Karl Popper: "Non-reproducible single occurrences are of no significance to science". Reproducibility assures that an effect is not observed due to chance or an experimental artifact resulting in a one-time event. Claerbout and Karrenbach define reproducibility in computational sciences, picturing it as a nametag attached to every figure caption in a manuscript, that can be used to recalculate the figures from all the data, parameters and programs. Between 1990 and 2006, people began to use terms like replication, replicable, replicability, ideally referring to the process of completely re-running an experiment, with all the permutations of new researchers, new equipment, new subjects or other raw materials.

Table 4.1: *Timeline of the interest in reproducibility and leading publications.*

1992	Claerbout and Karrenbach [28].
2004	Taverna [95].
2005	Galaxy [50].
2006	GenePattern 2.0 [108].
2006	Kepler [86].
2008	VisTrails [117].
2008	KNIME [16].
2009	LeVeque [82].
2009	Ioannidis et al. [74].
2009	Drummond [38].
2009	Vieland [133].
2010	Casadevall and Fang [24].
2011	Stodden [126].
2011	Peng [100].
2011	Prinz et al. [104].
2011	Jasny et al. [75].
2012	Freire and Silva [43].
2013	DataMill [33].
2014	Collberg et al. [29].
2014	Basel's Reproducibility Levels [55].
2014	Sumatra [31].
2015	PROVA! [87].
2015	Topalidou et al. [129].
2015	Liberman [83].
2016	ACM's 3R [41].
2017	PopperCI [76].
2018	Guerrera et al. [59].
2018	Barba [13].

An outlook of the terminologies used for Reproducible Research is given by [13], trying to organize all the words used by different research groups or communities while referring to the broad topic of reproducibility.

The value and importance of reproducibility have been demonstrated by studies showing impossibility or difficulty to replicate published results [74], failed clinical trials [104] [14]. The problem has been detected, and early results have been reported. Dolfi et al. [36] propose a model for reproducible papers such that "the current manuscript already contains sufficient details, codes, and scripts to reproduce all the presented numerical results and figures." A constellation of tools and prototypes is available, in the bioinformatics field, not only to document the workflow of an application but also to make it reproducible. Taverna [95, 98] and VisTrails [117, 43] integrate data acquisition, derivation, analysis, and visualization as executable components throughout the scientific exploration process. Repeatability is facilitated by ensuring that the organizations that design the workflows control the evolution of the software components used in them. This "controlled services" approach is shared by other WFMS such as Kepler [86, 50], GenePattern [108], Sumatra [31], and KNIME [16], an open-source workflow-based integration platform for data analytics.

Opening up access to the data and software, not just the final publication, is one of the goals of the open science movement, thus allowing to build upon existing work, either to test it or to develop new ideas. To do so, a correct sharing of data and code, together with a manuscript, may be beneficial.

Reproducibility of computer experiments demonstrates strength on the scientific point of view because a wider community not only can check the correctness of published results but is also able to compare different approaches. Experimental research is not time-independent: experiment parameters are changed over time with the goal of getting more precise results or better performance indices. Thus, systems that address reproducible experiments must support both present and past configuration settings in a flexible manner.

The major problem when talking about the reproducibility of experiments is that people other than the original researcher have to deal with the configuration of their environment, which likely differs from the one used in the original execution: it is therefore needed a dependency check, which must represent the first step of an experiment. A scientific discovery process may require the application of several steps and activities,

and it is necessary to trace and collect sufficient provenance information over this process, thus offering provenance support. Allowing to run tests automatically, makes it possible to self-document the environment in which the experiment was executed and store it.

The difficulty in reproducing computational research is in large part caused by the difficulty in capturing every last detail of the software and computing environment, which is what is needed to achieve reliable replication [32]. As can be found, articles often do not have a sufficiently detailed description of their experiments and do not make available the software used to obtain the results claimed. As a consequence, parallel computational results are sometimes impossible to reproduce, often questionable, and therefore of little or no scientific value [72]. In HPC, tools that focus on the reproducibility have been introduced only later, with PROVA! [87] and PopperCI [76].

The first step towards reproducibility in computational sciences is, therefore, a clear and precise description of the experiments: we propose a taxonomy of the computational experiments that should serve as a common denominator and basis.

4.1 Taxonomy of Experiments

Computational problem solving in general can be described as follows: "A computational **p**roblem is solved by an algorithmic **m**ethod on a compute **s**ystem". The triple *(Problem / Method / System)* represents a *micro-experiment*, which can be considered as being one point in the space of experiments (see Figure 4.1). An example of the description of an experiment in terms of *(Problem / Method / System)* follows:

- ***Problem***: Solve a (random) dense system of linear equations in IEEE double precision arithmetic.

- ***Method***: Two-dimensional block-cyclic data distribution. Right-looking variant of the LU factorization with row partial pivoting (see [5]).

- ***System***: Distributed-memory computer with Message Passing Interface (MPI 1.1 compliant) and Basic Linear Algebra Subprograms (BLAS) installed.

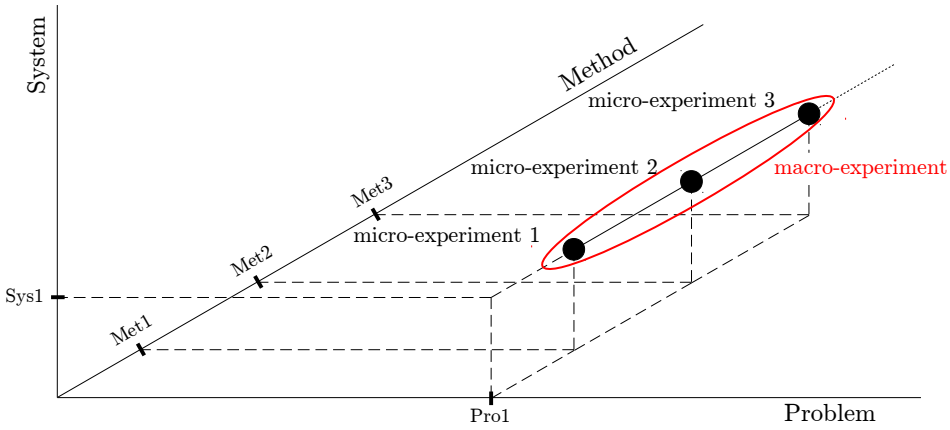

Figure 4.1: *Space of Computational Experiments*

Usually, to dive and understand a topic, it is needed to compare data resulting from more than a single micro-experiment. When keeping two out of the three dimensions fixed, the outcome is an experiment function of the third dimension: the red line shown in Figure 4.1 identifies such a *macro-experiment*, which is a collection of micro-experiments (e.g., the black dots in Figure 4.1). Macro-experiments can be categorized as being either system-oriented, method-oriented, or problem-oriented.

4.2 Reproducibility Levels

Once clarified the terminology to be used for computational experiments, it is possible to define levels of reproducibility expanding those concepts. The Association for Computing Machinery (ACM) has recently introduced the 3 Rs of the Research [41], and their description is pretty much overlapping our three levels of reproducibility, first introduced in Guerrera et al. [55], and evolved to the following version:

- **Repetition**: re-running the original *micro-* or *macro-experiment* without any variation of the parameters, should drive to the same results and a certain level of credibility is guaranteed (**completeness** of documentation)
- **Replication**: re-running an original experiment on a different system. An experiment should not be bound to a specific computing environment (**portability**)

- **Re-experimentation**: if changing the methods drives to the same outputs, the scientific approach is proven (**correctness** of the approach)

4.3 Goals of the PROVA! Project

Once defined the taxonomy of computational experiments, and the levels of reproducibility, the next step is to produce an effort to share them to the community and provide a reusable solution that can be adopted to create a collaborative ecosystem for the discipline of performance engineering (see Figure 4.2).

Performance modeling and experimentation are two interrelated tasks: once chosen a problem to solve, it needs to be studied and characterized by performance models. The understanding given by a model must be tested through experiments on selected systems, using specific methods. Ideally, the prediction of a model and the results of reproducible experiments match, thus the study can be trusted, and it is ready to be published and used by third parties in external applications. In case of a mismatch between models and experiments, it is crucial to analyze the results and understand whether the model needs to be adapted or the method used (and, talking about computational experiments in HPC, its code) requires an optimization.

In this work, we restrict our analysis to stencil problems, described in Chapter 7: different classes and kinds of stencils, are involved in the study and their performance (on several architectures), is predicted and measured.

The usability of the proposed framework is proved by using stencils as an input to our tool PROVA! (detailed in Chapter 5), that allows the user to repeat, replicate and re-experiment stencil computations. Repeating a stencil computation on a different architecture is complicated and subject to the knowledge of the configurations and parameters tuned in the original experiment. When someone else, other than the original author of the study, tries to repeat a stencil computation (even on the same system), several challenges emerge: completeness of experiment descriptions, portability, and reproducibility. Such problems are overcome when following the guidelines we introduce in Section 4.1 or using PROVA!. The performance results obtained using PROVA! can thus be trusted and compared with the predictions of the performance models.

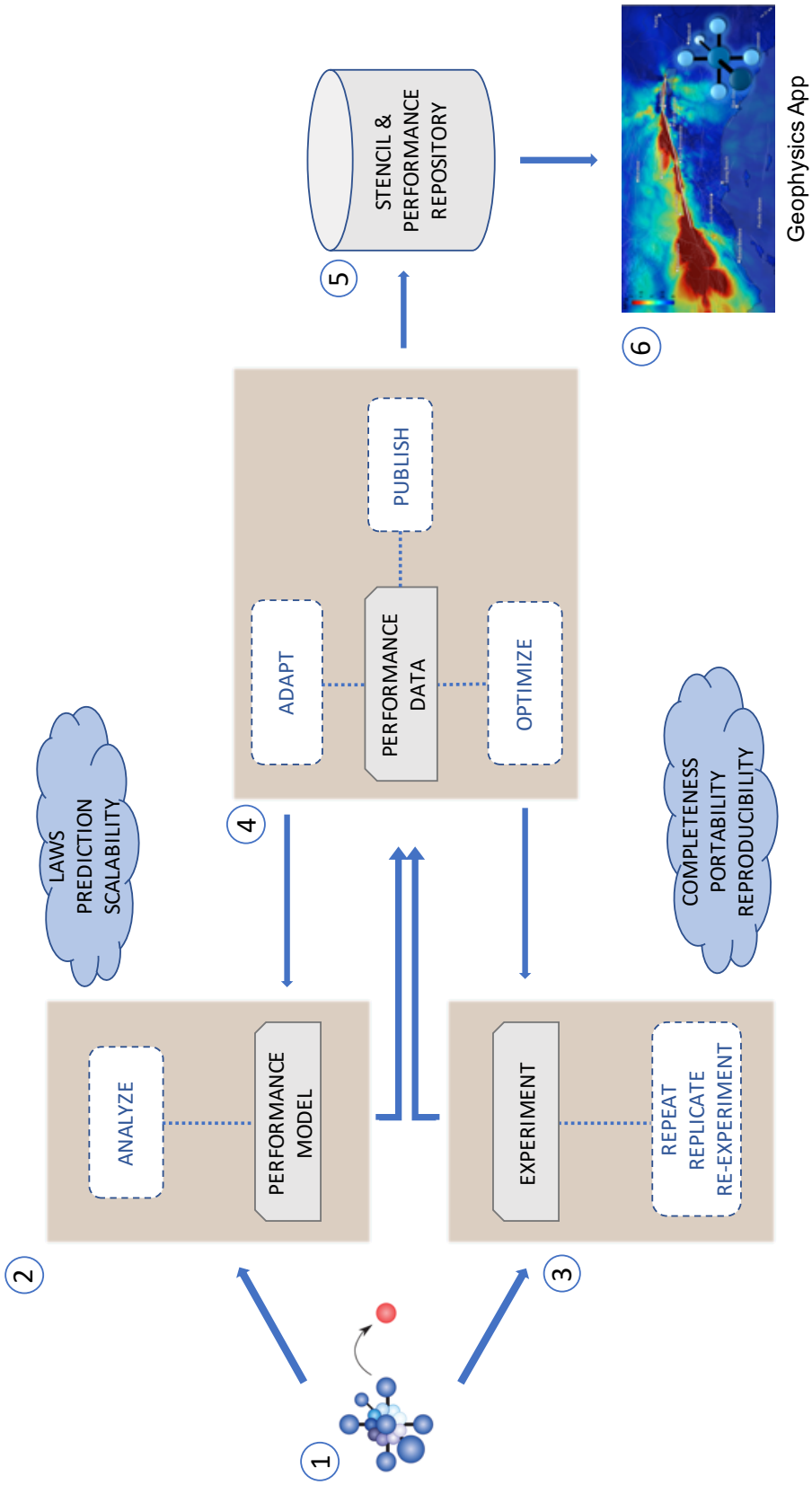

Figure 4.2: *Ecosystem for reproducible stencil experiments.*

Quantifying the theoretical performance of an HPC architecture is quite tricky due to the increasing complexity of both hardware and software characteristics, as shown in Chapter 2 and Chapter 3. Subsequently, it is hard to appreciate the gap between theoretical and observed performance: therefore we need performance models and tools to drive the process. Modeling techniques are essentials to understand the performance of modern architectures: models can show the bounds to the performance of a particular computation on a given architecture and help to understand where it is worth to optimize, by showing the bottlenecks of the target machine, thus guiding the optimization phase.

4.3.1 Contributions to the Project

The PROVA! project represents a joint effort carried out together with Antonio Maffia. The taxonomy of experiments and reproducibility levels have been defined and refined with his help. During the development of the PROVA! tool (detailed in Chapter 5), my focus has been on the management of the software environment (whose challenges have been described in Chapter 3) through modules and scientific software management tools, and the execution of the jobs on a parallel machine (whose challenges have been detailed in Chapter 2). His focus has been on designing containerization solutions to manage the software environment, gathering the output data and generating of performance graphs. The resulting tool can interface with an analysis and visualization server, that have been designed and implemented without my active involvement. The design and development of the stencil kernels generator (STEMPEL), its integration with PROVA! and the modeling tool (KERNCRAFT), as well as the experiments, have been carried out exclusively by myself.

Chapter 5

Overview of the PROVA! Tool

The aim of the PROVA! tool is to hide the complexity of the environment and the maintenance of the software stack, store valuable information about the system on which an experiment is performed, transparently to the users, thus making experiments reproducible without adding overhead. Since the tool automatically manages all the dependencies needed on the software level (i.e., its context), the users can focus on their research activity and without having to address reproducibility directly.

```
1:  $ workflow run_exp -p 2d-1risotropic-constant-star \
2:   -e 10 -d "3000 3000 200" -m openMP \
3:   -t 1 2 4 6 8 10 16 20 --pin node
```

Listing 5.1: *A possible command in* PROVA! *to run an experiment involving a 2-dimensional isotropic constant star stencil with grid size of* 3000^2 *using several thread counts and core affinity by node.*

The command shown in Listing 5.1 is one such command that may have used to run the experiments described in Chapter 10.

The aim of PROVA! is to provide an easy way to configure and run an experiment without dealing with the execution environment and then visualize the results in a meaningful way. As explained in Section 4.1, a *micro-experiment* can be thought as a triple $\langle Problem, Method, System \rangle$: our tool has been built upon such a definition. The first step to solve a problem, is the creation of a new project. Afterwards, methods, representing solutions (implementations), should be added to the previously defined project. The last step would be the execution of an experiment by running the solutions implemented.

Furthermore, PROVA! allows, after a configuration step, to run the same project on a different system.

5.1 Addressing Complexity

PROVA! is a distributed workflow and a system management tool that allows users to test several algorithmic methods on several systems quickly. That means, not only comparing different solutions but also revealing details and insights about the test systems.

Indeed, it is essential to test the code written or produced by an optimizing compiler on different machines. Tool developers usually focus and publish performance results on a specific machine (the one they have access to), often ignoring or not correctly addressing other architectures. It is worth noting that, even when developers treat several architectures, there are no studies we are aware of, which correlate methods and systems, meant as a combination of hardware and software.

While sharing the source code is beneficial, its availability does not represent a sufficient condition for reproducibility. In fact, the code may not compile, and even if it does, the results could be affected by the differences of other components in the software stack. Additional information like version of the compiler, compilation flags, configurations, experiment parameters, and raw results are fundamental contextual information for the reproducibility of an experiment. All these information are automatically stored by PROVA! when creating a project, a method, and an experiment. Each of them holds a descriptor that stores the relevant information and makes it available to the tool and the users when needed. Examples of these files are available at the repository [58], for the methods, the experiments, and the problems discussed in [59]. For each used software PROVA! stores the building and installing recipes, in the form of easyconfig files (used by EasyBuild [48, 70, 69]), the compilation and execution commands, together with their environment (automatically using modules, in a way transparent to the user), self-documenting the whole research from the creation of a project until the execution of an experiment. All the configuration files are stored and could be used by an independent researcher to recreate the state of the system at each step, even without using PROVA!, to reproduce an experiment.

The complexity that the architectures have reached has been already discussed in Chapter 2. Such a complexity translates into a vast potential

in terms of performance, at the cost of an architecture-specific tuning of the high performance codes. The tuning is required at several levels: in the source code of an application, at compilation time and also at runtime (in some cases). It is therefore required to meticulously note these details and share them to let the results be reproducible. PROVA! tries to fulfill this task by separating the phases of writing, compiling and running a code. There are meta-data that bind a particular code to an experiment, to the way it has been compiled, to the commands executed to run it and the environment available at that time, thus self-documenting the whole life-cycle of an experiment.

5.1.1 Implementation Phase

In the HPC field, an experiment is often a code that fulfills some requirements and tries to solve a well-defined problem. The availability of source code of an experiment is a pre-requisites for the reproducibility of the research, thus requiring to be stored together with the results of an experiment and possibly shared.

5.1.2 Compilation Phase

One of the important phases identified is the compilation phase: as clarified in Section 3.3, optimizations, and modifications are carried out in this phase, that affect the performance of the resulting machine code. Consequently, for pursuing reproducibility, it is necessary to keep track of the flags used by the compiler and any other software package available and needed. It is particularly valid in case of stencil compilers, which accept several possible optimization requests on the command line.

5.1.3 Execution Phase

Not only the compilation can affect the results of an experiment, but also the way a certain execution is spawned. Scientific software packages often accept a myriad of command line parameters, not only directly related to the code that is going to be executed but also affecting the environment or configuring the host in a specific way. It is the case of MPI codes, where requests regarding the global and local options, the mapping of processes to resources, and their distribution can be done via command line arguments. HPC systems usually have a job scheduler

that accepts booking of requests, through so-called job files, which also need to be stored and attached to an experiment.

To analyze and interpret the results, it is crucial to trust and to be able to reproduce them. PROVA! manages the software stack, but other parameters can affect the measurements, such as the way the threads are assigned to the available resources, as described in Chapter 10. As stated in [130, 113], thread/process affinity is vital for performance. Correct pinning is even more critical on processors supporting SMT, where hardware threads share resources on a single core. With Likwid [130] no code changes are required, and it offers a portable approach to the pinning problem. This motivates the integration of Likwid[112] inside PROVA!: the explicit pinning of the threads to the cores (processes) is wholly delegated to it. Different options for the pinning have been implemented, that allow to dynamically build a the likwid call, transparently to the user, as described in details in Section 6.3.

5.2 Walkthrough

In this section, we show how an experiment can be created and executed, starting with the definition of the problem.

The first step when using PROVA! is the creation a project, i.e., a problem and its characteristic parameters. This is done through the command shown in Listing 5.2, that creates a project called *KNL*, whose parameters are *X_MAX Y_MAX Z_MAX* and their default values 100 100 100.

```
1:  $ source /export/hpwc/PROVA/util/BaseSetup.sh
2:  $ workflow project -c -p KNL --params "X_MAX Y_MAX \
3:  Z_MAX" --values "100 100 100" --threads 64
```

Listing 5.2: *Command used in* PROVA! *to create a project named KNL having X_MAX, Y_MAX, Z_MAX as parameters and default values of* 100.

Additionally, since HPC machines are the target, a default amount of threads must be defined. Such a command line, simply represent a description of the relevant characteristic of a problem. An optional comment can be added, textually describing the project. This information is stored as metadata of the project and later used to retrieve it.

The next step is the creation of a method, i.e., an implementation of an algorithmic way to solve the problem we are targeting. Usually, some software packages are required for a solution to execute. Said packages

are passed as an argument to the method creation routine, as shown in Listing 5.3.

```
1:      $ workflow method -c -p KNL -m \
2:      OpenMP-4.0-GCC-4.9.3-2.25 -n wave
```

Listing 5.3: *Command used in* PROVA! *to create an implementation named wave (of type OpenMP-4.0-GCC-4.9.3-2.25) and belonging to the project KNL.*

In this case, the method *wave* is created and added to the previously created project. The *methodType*, namely the collection of software packages required to run the freshly created method, is *OpenMP-4.0-GCC-4.9.3-2.25* : its name reflects the naming scheme used by EasyBuild. Such a method type defines not only all the environment modules (installed through EasyBuild and loaded through Lmod) needed to run a method but also contains scripts describing how such software must be compiled and executed. PROVA! offers some *methodTypes*, which fulfill general needs in performance benchmarking experiments. The expert user can define new *methodTypes* on his own or ask an admin to do it. The creation of a new *methodType*, requires the knowledge of both PROVA! and EasyBuild. An example of that is shown in Appendix B.

The result of the command presented in shown in Listing 5.3 is the creation in the project folder, of a directory, representing the method, and containing subfolders supposed to contain source code, executable and outputs. In the *src* folder, is present a Makefile that allows customizing the compilation. In the case illustrated, the *methodType* requires only a compiler, GCC version 4.9.3, compiled using binutils version 2.25. The related compilation script, not modifiable by the user, contains merely a call to *make* the source code. The customization of the Makefile, as previously mentioned, is left to the user.

After having created a method inside a project, the user can start implementing his solution into the source code. Once the solution is complete, it is possible to compile it by using the command line shown in Listing 5.4, that forces the compilation of the method wave of the project KNL.

```
$ workflow compile -p KNL -n wave
```

Listing 5.4: *Command used in* PROVA! *to compile the implementation named wave and belonging to the project KNL.*

If the code compiles and it is the actual solution one wanted to implement, then its runtime characteristics can be tested by running it with the default parameters, as shown in Listing 5.5.

```
$ workflow run -p KNL -n wave --pin none
```

Listing 5.5: *Command used in* PROVA! *to execute with default parameters and no explicit pinning the implementation named wave and belonging to the project KNL.*

Once checked the output and confirmed that the semantics of the code is correct, it is possible to create an experiment. The experiment descriptor contains information that uniquely identifies an experiment and all the setup involved, e.g., methods used (and therefore their *methodTypes*), project executed, default and runtime parameters. The way a certain method is run is defined by the *methodType*. In the case of a simple OpenMP code, like the method we just defined, it simply contains a call to the binary file created, to which is pre-pended the likwid command-line needed for the pinning of the threads to the processors, as shown in Listing 5.6.

```
$ workflow run_exp -p KNL -e 5 -d "200 200 200" \
"150 120 120" -m wave -t 2 4 8 16 32 64 128 256 \
--pin none
```

Listing 5.6: *Command used in* PROVA! *to execute an experiment with non-default parameters and no explicit pinning. A list of of threads to use and input parameters is passed on the command line.*

Several values for the parameters are accepted, and will result in several *micro-experiments* being run.

Since HPC systems usually use a job scheduler to accept executions, an interface to several types of job scheduler has been defined and implemented. The execution of an experiment through a job is done running the command shown in Listing 5.7.

```
$ workflow job_run_exp "pbs" "1" "smp" "00:59:00" "1" \
-p KNL -e 5 -d "200 200 200" -m wave -t 2 4 8 16 \
32 64 128 256 --pin node
```

Listing 5.7: *Command used in* PROVA! *to execute an experiment with non-default parameters and explicit pinning by node. The experiment is submitted through the job scheduler PBS. A list of of threads to use and input parameters is passed on the command line.*

Compared to the command in Listing 5.6, there are some extra parameters at the beginning that identify the type of job scheduler to use (in this case PBS), the number of nodes to use (in this case, being the method a plain OpenMP solution, only 1 node is required), the partition of the cluster to use, the wall-clock time to reserve for the job and the memory required.

Once the experiment successfully terminates, it is possible to retrieve the performance data and automatically plot it into a histogram of the chosen metric, as shown in Listing 5.8.

```
$ workflow build_graph -p KNL -e 20171129_140354 \
-d "200 200 200" -m wave_none -t 2 4 8 16 32 64 128 \
256 -f 0 -T stdev -M mlup/s
```

Listing 5.8: *Command used in* PROVA! *to generate an histogram of the performance output obtained in the experiment named 20171129_140354. Additional parameters restrict the data to be visualized by specifying the number of threads, the metric to visualize and the error bar.*

Information required is the name of the project and the identifier of the experiment (a string with the date of execution), the value of the parameters we are interested into, the kind of error-bar desired, and the metric.

Chapter 6

Implementation Aspects

6.1 Architecture

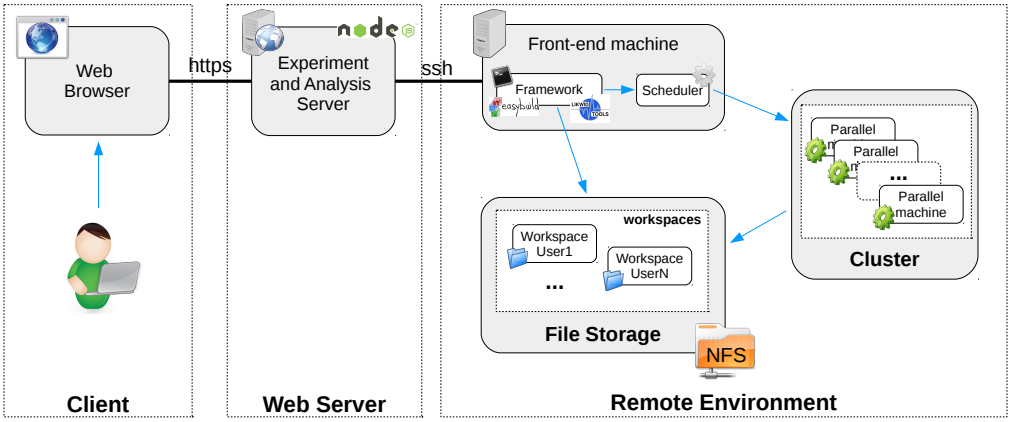

Figure 6.1: *Architectural overview of* PROVA!.

PROVA!, developed for a Unix environment, is mainly composed of two entities: the framework itself, which interacts with the parallel machines, and a web interface. In this work, we focus on the framework. It represents the core of our system and should be installed system-wide by an administrator, or locally by a user, on the front-end of every cluster on which it has to be used: in this way it can manage the software stack of the whole machine. The consistency of the software stack is ensured by using EasyBuild [48, 70], a scientific software management, and installation tool, currently used in major HPC centers across Europe (Juelich SC, CSCS Lugano, VSC Ghent).

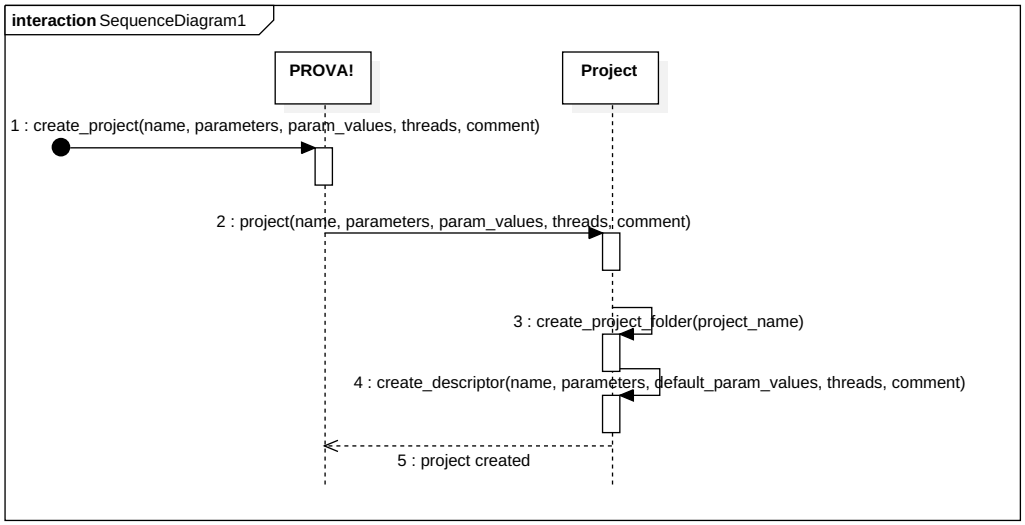

Figure 6.2: *UML diagram schematizing the creation of a project in a PROVA! workspace.*

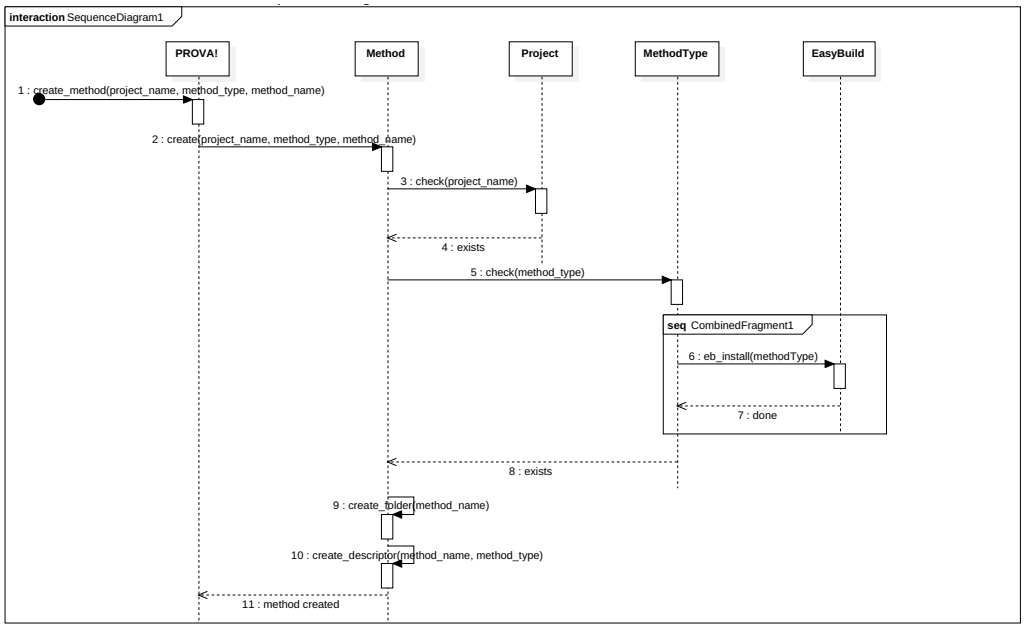

Figure 6.3: *UML diagram schematizing the creation of a method in a project.*

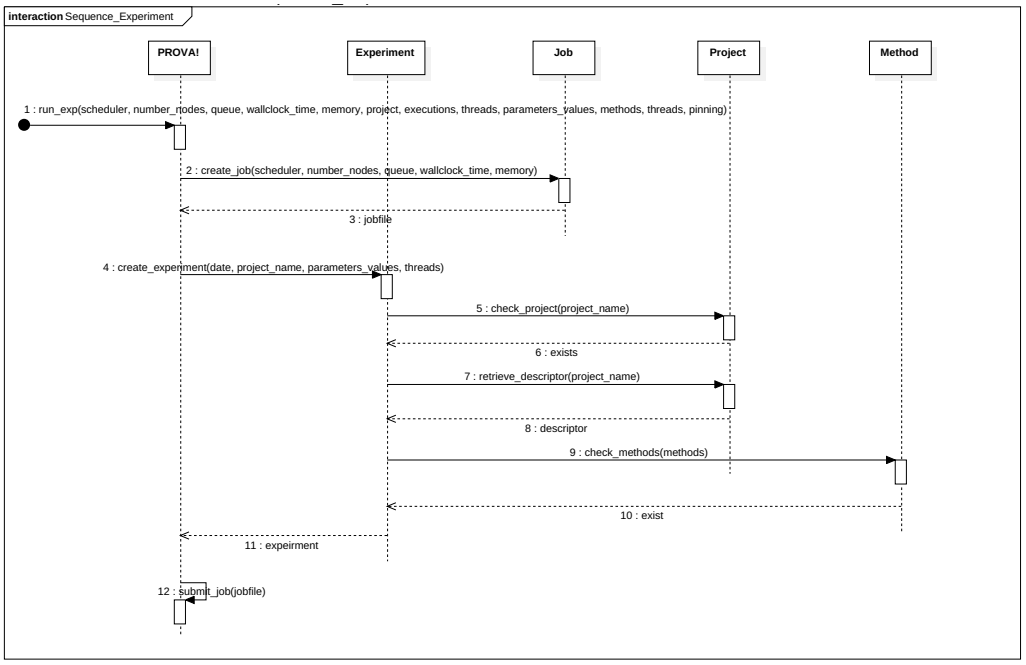

Figure 6.4: *UML diagram schematizing the creation of an experiment to run the methods created in a project.*

The core of PROVA! consists of a collection of bash and python scripts, offering the possibility to install modules (method types), to create and manage projects and methods (Figure 6.2 and Figure 6.3), as well as to execute experiments (Figure 6.4) and visualize the output (Figure 6.5).

A first prototype of the tool has been presented in Maffia et al. [87]. Compared to the version described there, PROVA! has been extended with the possibility to interface with a job scheduling system (see Figure 6.1): scientists use the tool either via command line or through a web browser, accessing the remote machine on which they want to execute the experiments. The connections from the web server to the remote machine consist of *SSH accesses*. The core framework installed on the login node of a cluster is used to locally manage projects (and the related software and dependencies) and experiments.

Figures 6.2, 6.3, 6.4, and 6.5 describe the sequence of calls that are needed to create a new project, create a new method (when the *methodTypes* needed are already available in the system), create and run an experiment (after having implemented the source code to run), and generate the performance graph after a successful execution of an experiment.

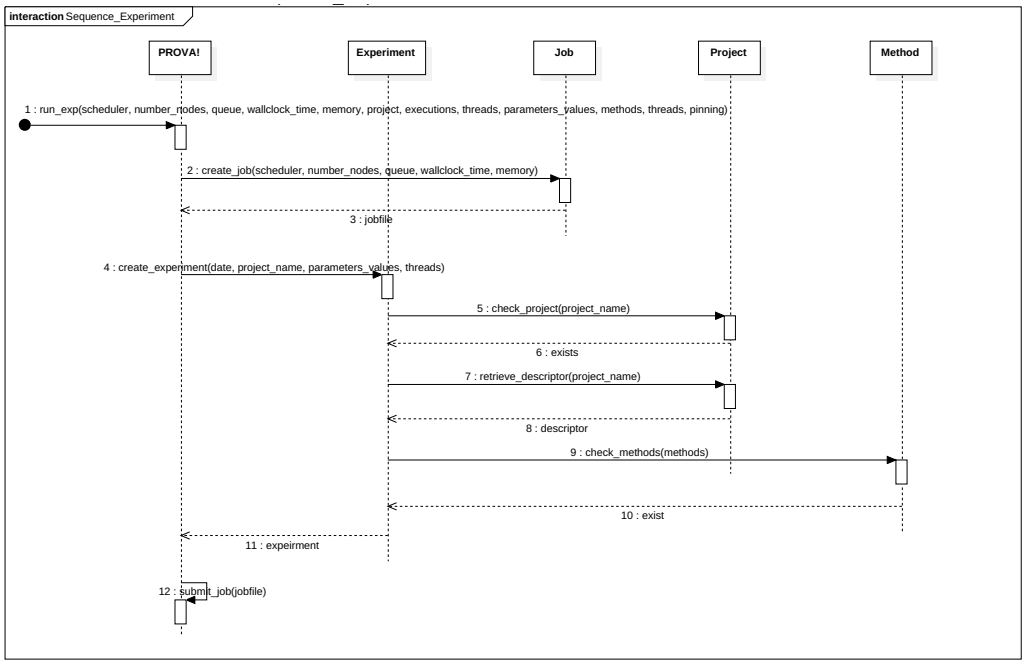

Figure 6.5: *UML diagram schematizing the creation of a graph showing the results of the execution of an experiment.*

6.2 Mapping of the Experiment Taxonomy

The taxonomy of an experiment, introduced in Section 4.1, is a critical factor for its reproducibility. Three are the main factors that have been identified as responsible for the reproducibility or irreproducibility of an experiment: system, problem, and method. Consequently, they all have abstractions into the software level. Since a user has no control on the hardware, the hardware characteristics of a system are merely stored by PROVA!, so that they will always be available, when it comes to interpreting the results of an execution or in case of discrepancies while replicating the experiment, thus helping to address the root of the issue. The software characteristics of a system instead, are partially mapped in the concept of *methodType*, described in Section 6.2.2.

6.2.1 Projects

The abstraction of a problem is called project. It is meant to store the information related to the problem one may want to solve, together with the parameters introduced. The information is saved in a file called project descriptor: it stores not only the project name, but also its default param-

eters, and a string serving as a description. A project is a container for the methods used (and implemented) to solve a specific problem.

6.2.2 Methods

A method is defined as the algorithmic way to solve a problem. In computational sciences, at the lowest level, it is a code, written in any programming language, that while running on a system, tries to find a solution for the problem/project it refers to. Since it must run on a computer system, it needs to go through several stages, from being written and tested, to be compiled and then executed.

The complexity related to its execution can vary from the basic need of a compiler (e.g., GCC or Intel compiler) to the need of a complete chain of software that must be installed and present in the environment of the system while the code is run. For instance, in computational molecular dynamics, scientists often use a software called GROMACS to run their experiments. In such a scenario, GROMACS must be available when running a simulation. All the software needed by a certain method are registered and stored in a method descriptor, together with the method name and a short description. The software entries in such a descriptor represent software modules, by default Lmod modules. For each used software, PROVA! stores the building and installing recipes, in the form of *easyconfig* files, used by EasyBuild [48, 70, 69].

Between the abstraction of Method, namely the code one wants to run, and the software needed for it to run, namely a Lmod module, there is the abstraction of *methodType*. It binds the method, the required software counterpart and they way the user code must be compiled and executed (in the form of compilation and run scripts).

Acting this way, the whole research from the creation of a project until the execution of an experiment is self-documented. All the configuration files are stored and could be used by an independent researcher to recreate the state of the system at each step, even without using PROVA!, to reproduce an experiment.

6.3 Likwid Interface

To analyze and interpret the results, it is crucial to trust and to be able to reproduce them. PROVA! manages the software stack, but other pa-

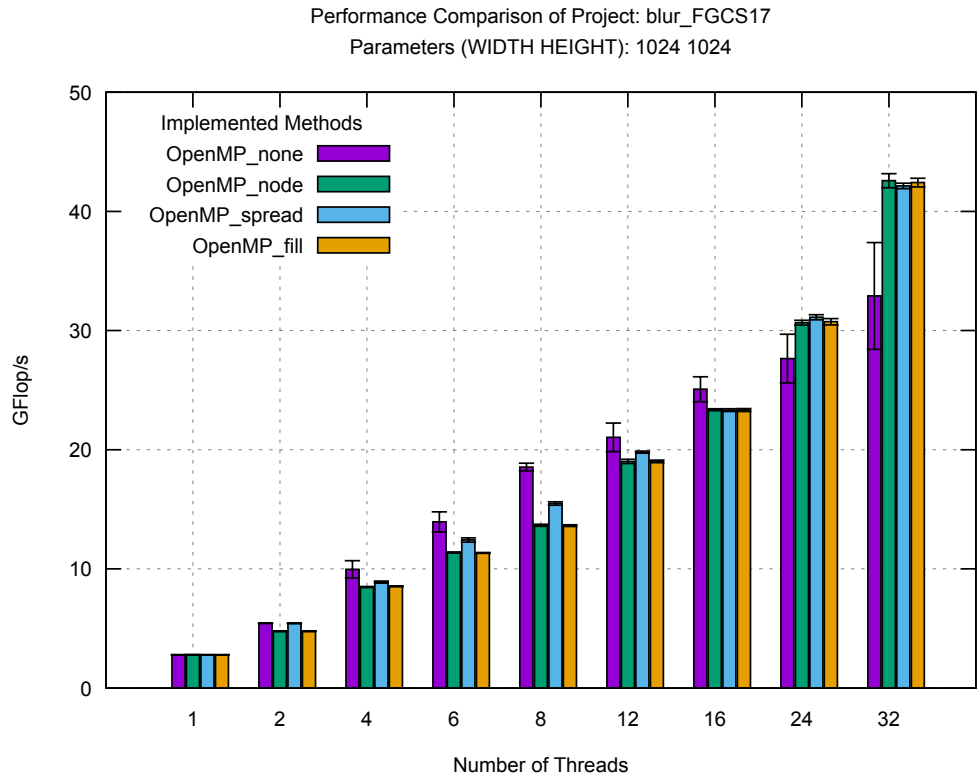

Figure 6.6: *Performance graph of a 2D Gaussian blur, with naive OpenMP implementation both with and without explicit pinning on Mint, taken from [59]. The histogram shows the average value out of 5 executions, and the error bars the standard deviation.*

rameters can affect the measurements, such as the way the threads are assigned to the available resources.

As stated in [130], and described in Section 2.3.1, thread/process affinity is vital for performance. Correct pinning is even more important on processors supporting SMT, where hardware threads share resources on a single core. With Likwid [130] no code changes are required, and it offers a portable approach to the pinning problem. Thus, Likwid[112] has been integrated into PROVA!, delegating to it the explicit pinning of the threads to the cores. The pinning is performed through three possible strategies, nominally: *ByNode*, *ByFilling*, *BySpreading*. Said strategies exploit the abstractions of logical nodes and locality domains (in particular the socket), offered by likwid (when present at the hardware level).

In Section 9.1.3 and Section 9.1.4, is shown the output of *likwid-topology*, which graphically presents the architectural details of a compute node.

The cores available on a socket have a *physical* and *logical id* that do not always match. The pinning strategy *ByNode* binds a thread to each of the logical nodes, by increasing *id*. In the computer systems available at the time of writing, a node is usually composed of two or more sockets, hosting each a microprocessor. Even inside a single microprocessor, as shown in Section 2.3.1, several locality domains can be present. The pinning strategy *ByFilling* binds the threads to the processors, taking care of assigning first to the processors in a single socket and then, when each processor on the first socket has received its thread, moving to the next available socket. The strategy *BySpreading* follows a complementary approach: it tries to balance the threads over the available sockets, assigning threads in a way that equally distributes them to the sockets. Listing 6.1 shows an example of how each of the three strategies, translates into a likwid command, in case of executing *mycode* with 4 threads.

```
# byNode
likwid-pin -c N:0-3 ./mycode

#byFilling
likwid-pin -c S0:0-3 ./mycode

#bySpreading
likwid-pin -c S0:0-1@S1:0-1 ./mycode
```

Listing 6.1: *Example of how the pinning strategies defined by* PROVA! *translate into a likwid command.*

Figure 6.6, taken from [59], presents a test case of a 2-dimensional Gaussian blur applied to a 1024x1024 grid, containing values in single precision (float): all the histograms represent the performance of the same code, where merely the pinning strategy has been varied, using no explicit pinning (suffix *_none*), and pinning using the previously described strategies. The testbed is the Mint cluster at the University of Basel, whose nodes are dual socket AMD Opteron 6274 "Bulldozer" with a nominal clock speed of 2.2 GHz and 16 cores per chip.

When the goal is to characterize different systems, one cannot rely on the OS for managing the threads but must define the thread/core affinity explicitly.

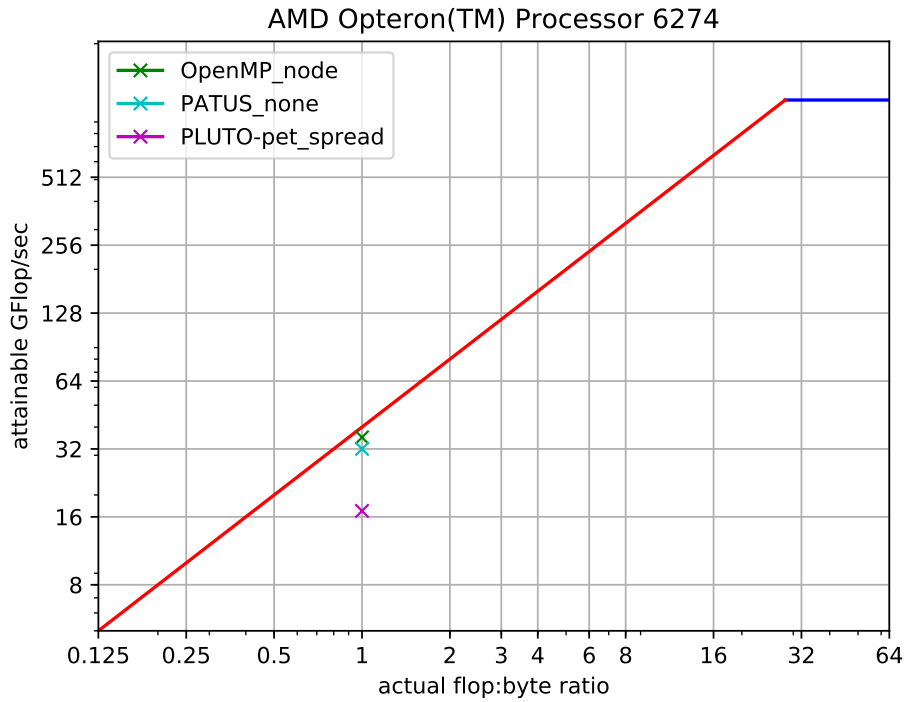

Figure 6.7: *Roofline for the Mint cluster with the three kernels implementing a 3D wave equation. The description of the experiment and the discussion of the results have been published in [59].*

6.4 Empirical Roofline

The Roofline model requires a manual analysis of the architecture and the kernels that will populate the roofline, defined as:

$$Attainable\ GFlop/sec = min(P_{max}, B_s * AI),$$

where P_{max} is the peak floating point performance (representing a hardware limit), B_s is the memory bandwidth, and AI is the arithmetic intensity.

While B_s can be measured using the STREAM benchmark [92], P_{max} must be calculated as:

$$P_{max} = cores * frequency * FLOPS\ per\ cycle,$$

where *cores* represents the number of real cores available on an architecture, $frequency$ is the clock frequency used by the machine and $FLOP\ per\ cycle$ (taken from the architectural specification of a machine) represents the

number of floating point operations that said architecture could carry out over a clock cycle.

The roofline can be plotted and visually inspected for a certain architecture. On the plot, can then be added the kernel under investigation: the AI of a kernel lays on the x-axis of the graph and the *Attainable GFlop/sec* on the y-axis.

In Figure 6.7 is shown an example of a roofline plot for a 3D wave equation implemented using three different methods: naive OpenMP implementation, PATUS, and PLUTO, presented in [59].

At the time of writing, PROVA! has only basics automated roofline capabilities, which should be extended. The arithmetic intensity and the floating point operations must be calculated by hand and passed to the tool as input: in future, all necessary data will be automatically calculated using values from the hardware counters or Kerncraft [64].

Part III

Experimental Evaluation

Chapter 7

Parallel Stencil Codes

7.1 Motifs

In his 2004 lecture titled *Defining Software Requirements for Scientific Computing*, Phillip Colella identified the so-called *seven dwarfs*, which are defined as algorithmic methods that capture a reusable pattern of computation and communication. He identified and presented them because he was convinced they would be of extreme importance for the numerics research for at least the next decade. The dwarf idea was later taken up by scientists at Berkeley and the list of dwarfs (renamed to motifs) was extended [11]: structured grids, unstructured grids, spectral methods (FFT), dense linear algebra, sparse linear algebra, n-body methods, montecarlo/map reduce, combinational logic, graph traversal, dynamic programming, backtrack and branch-and-bound, graphical models, and finite state machines.

Stencil computations are present in said list as motif *Structured Grid*. A stencil defines operations on a multi-dimensional grid, which are repeatedly applied such that the value of a grid point depends on the value of the point itself and the ones of its neighbors, until a certain range or radius, in a previous time step. The values of the neighborhood can be weighted, by some coefficients.

Stencil structures usually occur when applying a discretization to differential operators. A sweep is the application of a stencil operator to all the points of a grid. It is possible that both the input and output grids of a stencil computation are the same, in which case the order of traversal of the points matters since the method defines which points must use the

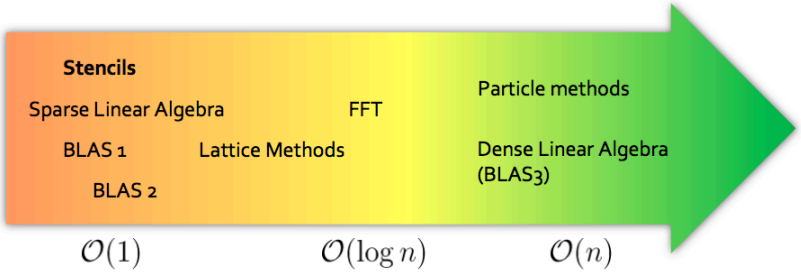

Figure 7.1: *Arithmetic intensity in various computational codes.*

updated values and those that still use the old values. Such a scheme of traversal is called Gauss-Seidel iteration. The Jacobi method instead uses two different grids as input and output. Thus, the traversal order is irrelevant. In this work, we restrict ourselves to Jacobi iterations.

While applying a stencil, it is important to define whether the points at the boundary are particular points that can be treated keeping the values constant (Dirichlet boundary conditions) or keeping the flux through the boundary constant (Neumann boundary conditions). In this work boundaries are not part of the computation.

The stencil motif has numerous applications in science and engineering such as weather forecast, geophysics, computational fluid dynamics, and image processing.

Stencil computations expose a high degree of parallelism. However, performance is not for free. They are memory-bound as typically only a limited amount of computation is performed per grid point, i.e., low arithmetic intensity. Arithmetic Intensity is the ratio of total floating-point operations to total data movement (bytes). Figure 7.1 shows how arithmetic intensity changes over some selected class of applications. Because of this memory bandwidth limitation, different optimization strategies can be applied to stencils. For example, if the application requires the stencil to be applied multiple times, there is potential to exploit temporal data locality, i.e., reuse cache data across iterations.

A fair amount of research has addressed the question of how temporal and spatial optimization can be done and what algorithmic changes and code transformations are needed. It is the purpose of our case study to experimentally explore different compilation methods. In Section 7.4 are presented two stencil compilers, that will be then used for the experiments.

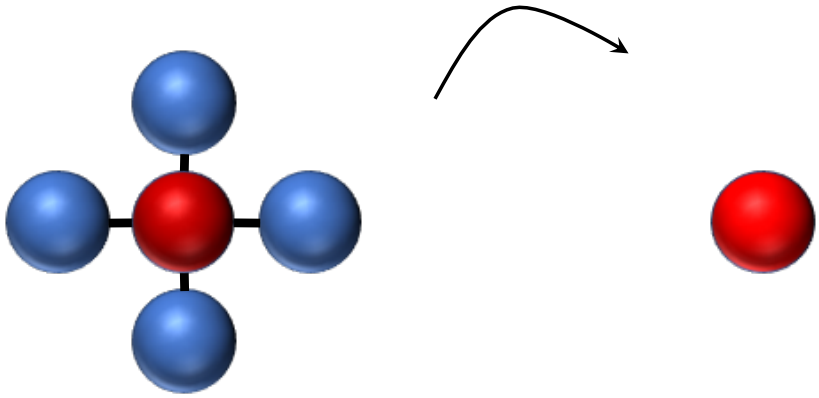

Figure 7.2: *2-dimensional star stencil with radius 1, constant and isotropic coefficients, yielding 6 FLOP.*

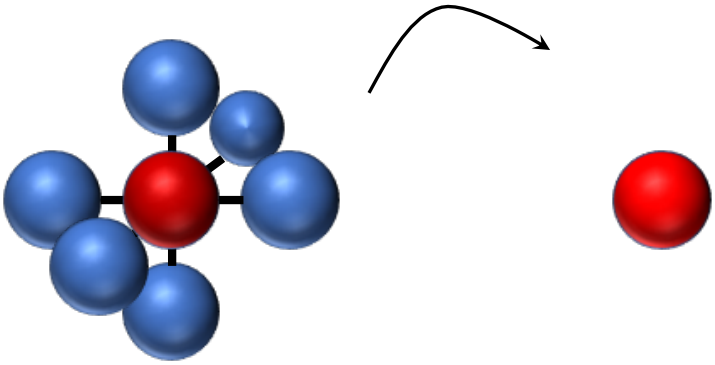

Figure 7.3: *3-dimensional star stencil with radius 1, constant and isotropic coefficients, yielding 8 FLOP.*

7.2 Stencils Classification

Stencil computations present a variety of patterns, useful to characterize the stencils themselves: dimensionality, shape, radius (or range) symmetricity of the coefficients, type of coefficients. In addition to this intrinsic parameters, the data type used to represent the values of the grid points can impact on the performance of a selected stencil on a given architecture. Such a characterization of stencils emerged within the German Federal Ministry of Education and Research project SKAMPY [3, 134].

The characterization keywords are self-explaining. Dimensionality identifies the number of dimensions over which the stencil expands: they range from uni-dimensional to n-dimensional, but in scientific computations, the interesting values for the dimensionality are 2 and 3. In this

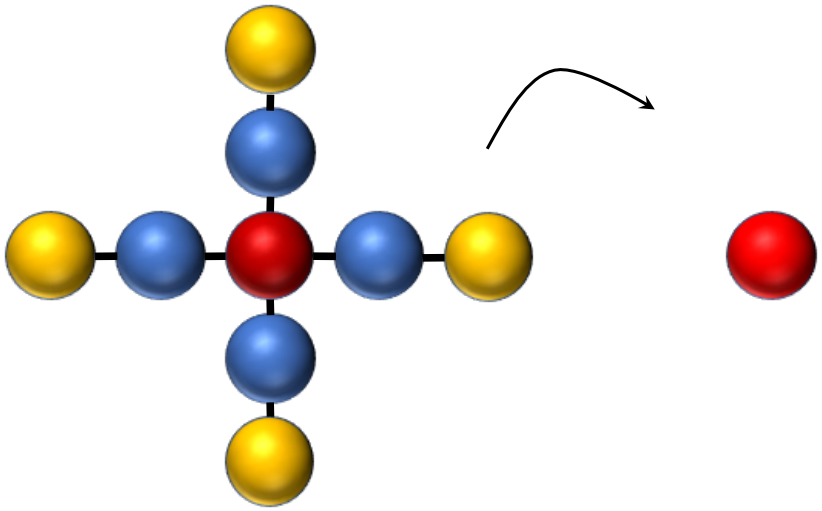

Figure 7.4: *2-dimensional star stencil with radius 2, constant and isotropic coefficients, yielding 11 FLOP.*

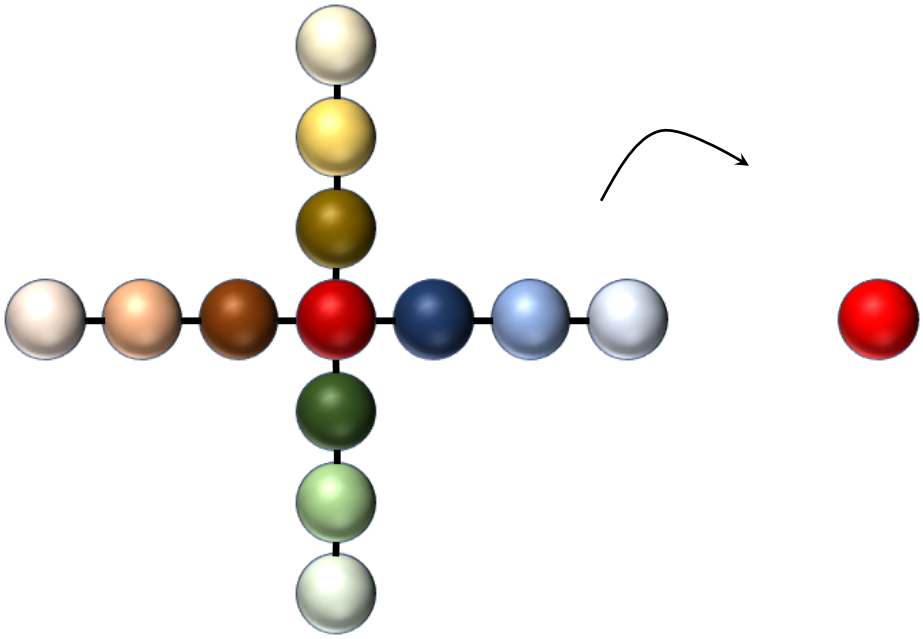

Figure 7.5: *2-dimensional star stencil with radius 3, constant and heterogeneous coefficients, yielding 25 FLOP.*

work, we restrict ourselves to those values, and examples are shown in Figure 7.2 and Figure 7.3.

The shape of the neighborhood involved in a stencil function can vary from star, to box, to hybrid combinations. In case of a star stencil (see Figure 7.2 and Figure 7.3) the points involved (the active neighborhood)

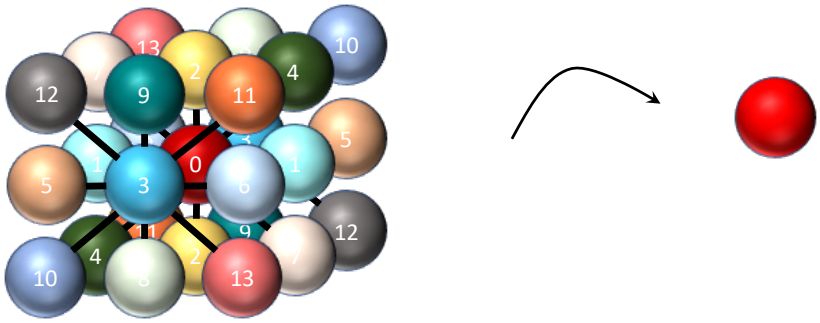

Figure 7.6: *3-dimensional box stencil with radius 1, variable and point-symmetric coefficients, yielding 40 FLOP.*

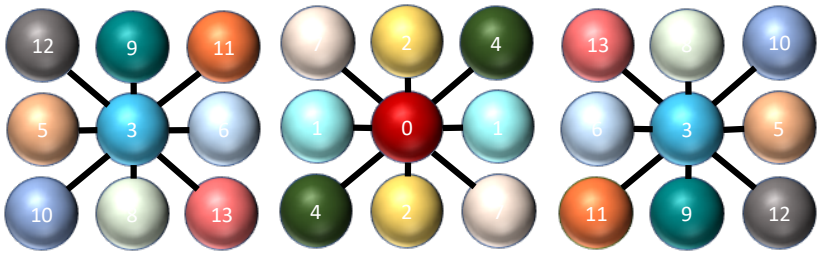

Figure 7.7: *Unravelling of a 3-dimensional, radius 1, box stencil with variable and point-symmetric coefficients in order to appreciate their symmetricity.*

are the one laying on the same axis as the center point. A box stencil (see Figure 7.6), instead, considers as neighbors of the center point, not only the ones laying on the same axis but all the ones that fall into a box having as perimeter the selected radius.

The radius represents the maximum distance from the center point, over an axis, to which a point is considered neighbor.

The values of the neighborhood can be weighted, by some coefficients. In case the coefficients are fixed for each of the points in the neighborhood, i.e., do not vary while the stencil operator moves over the grid, we talk of constant coefficients, whereas when each of the points of the grid has a different weight, we talk of variable coefficients.

To some extent, the weights applied to the points of the stencil can expose some symmetricity that helps to characterize the stencil as: homogeneous, heterogeneous, point-symmetric and isotropic. Homogeneity means that a single weight is applied to each point of the neighborhood, whereas a heterogeneous stencil presents a different coefficient for each point, as shown in Figure 7.5. Isotropicity happens when the coefficients vary with the radius, as shown in Figure 7.4. Point-symmetricity instead,

is the condition when the coefficients are symmetric to the center point of the stencil, as shown in Figure 7.6 and its unraveling, Figure 7.7, that allows appreciating the symmetricity of the coefficients.

The type of the coefficients can be constant in space and time or variable. Such differentiation is extremely relevant when modeling a stencil because, in the case of variable coefficients, the dominant working set can move from the grid points being updated to the coefficients. As described in [89], in the PDEs domain, the variability of the coefficients can stem from parameters, such as conductivities, that depend on the space or time.

In all the picture presented, the spheres represent points of a grid, whereas the red color identifies the center point of the stencil (the point of interested at the time) and all the other colors are used to represent the coefficients used to weight each point.

The number of FLOP varies according to shape, radius, symmetricity, and kind of coefficients. For each of the stencils shown in the Figures 7.2-7.7, the FLOP are specified.

7.3 Stencil TEMPlating Engineering Library

During this work arose a need for a representative set of stencil kernels. At the best of our knowledge, no such a set is publicly available, so that a Stencil TEMPlating Engineering Library (STEMPEL) has been developed.

Starting from the stencil classification introduced in the previous section, STEMPEL first generates a pseudo-C code that represents the stencil kernel, and then transforms it in compilable C code with OpenMP parallelization (and optionally with blocking), for benchmarking purposes. The final source code also contains a NUMA aware initialization of the arrays representing the grid on which the stencil operator is applied.

Furthermore, STEMPEL serves as an interface between the modeling of the performance of a stencil kernel and its reproducible execution. Figure 7.8 presents a schematization of its architecture.

An interface to Kerncraft, implemented in STEMPEL, allows obtaining a performance modeling. Said interface passes over to Kerncraft the pseudo-C code representing a kernel and an architecture descriptor. Additional and optional parameters are accepted to refine further the performance prediction required, such as the grid sizes. STEMPEL thus ex-

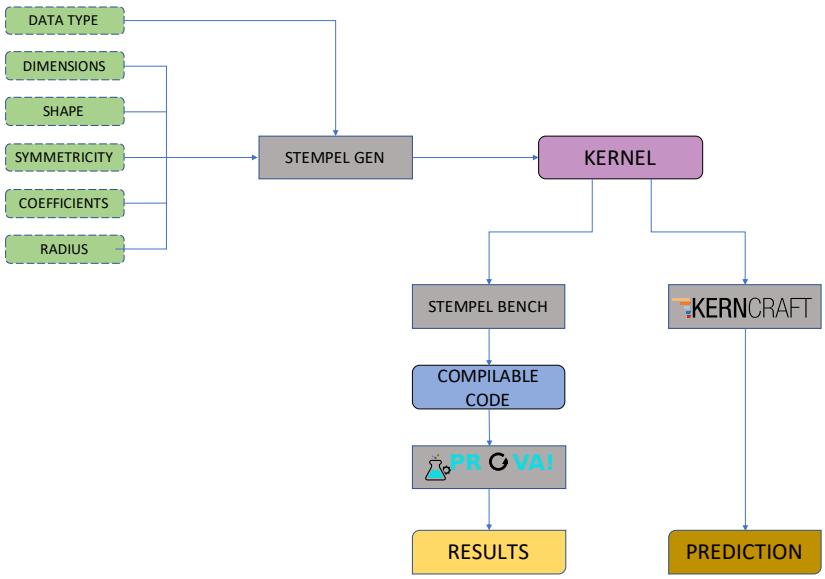

Figure 7.8: *Architecture of STEMPEL: the generated stencil kernel is passed to Kerncraft, for the performance modeling, and to the benchmark generator, that interfaces with* PROVA!.

ecutes a call to Kerncraft, requesting both the ECM and Roofline prediction for the kernel and the architecture specified.

Kerncraft offers the option to choose between two possible cache predictors, namely layer condition and cache simulation with pycachesim. STEMPEL adopts a fallback strategy, by usually requesting the layer condition analysis, and if that one fails, the cache simulation.

The prediction for the ECM model is obtained when using only one core and then multiplied either by the saturation value or the number of cores in the socket, whichever is smaller. Instead, the prediction for the Roofline model is obtained using the number of cores available in a socket. Once the stencil has been successfully modeled, the output is stored, and STEMPEL prepares the execution of an experiment through PROVA!.

As described in Chapter 5, the usage of PROVA! passes through the creation of a project (defining the problem to solve) and of a method, that implements a possible solution using specific software packages. STEMPEL provides an interface to PROVA! that is responsible for transparently setting up a project and creating a method in it. Such a method contains the compilable C code with OpenMP parallelization that is generated by STEMPEL, as described above. Afterward, the micro-experiments are executed, using the number of threads specified via the STEMPEL com-

mand line. It is also possible to customize the number of executions required. Once an experiment successfully terminates, the performance data are stored together with the predictions, thus allowing a comparative analysis.

It is worth noting that the calls to Kerncraft and PROVA! are entirely transparent to the user, who has partial control over them by optional parameters passed to STEMPEL.

7.4 Stencil Compilers

Because of their low arithmetic intensity, i.e., the small number of floating point operations per transferred data element, stencil computations typically are limited by the available bandwidth to the memory sub-system. As a consequence, it is essential to efficiently use the caches, by optimizing data locality (both spatial and temporal). Bandwidth-saving schemes like cache blocking techniques and methods to block across multiple time steps are used to increase the arithmetic intensity.

Hardware-aware programming techniques can also help: software prefetching, NUMA-aware data initialization, or cache bypassing represent a way to reduce bandwidth usage further.

A variety of solutions to this computational problem has appeared in literature, ranging from algorithmic approaches, like the cache-oblivious schema proposed by Frigo and Strumpen [44], the polyhedral model, auto-tuning approaches, and the usage of domain-specific languages.

7.4.1 PLUTO

PLUTO [18] is a source to source compiler that uses the polyhedral model approach for compiler optimization: applicable to loops with affine index functions and affine loop bounds, it interprets the iteration space as a polyhedron, and loop transformations correspond to operations or affine transformations of that polyhedron.

The compiler accepts C code as an input and provides as an output an OpenMP parallelized C code, with coarse-grained parallelism and data locality simultaneously. The final code is also optimized for locality and made amenable for auto-vectorization. It is also able to produce code exploiting the diamond tiling technique [17].

7.4.2 PATUS

PATUS [26] is based on a DSL + Auto-tuning approach, focusing on obtaining performance without affecting the productivity. Domain Specific Languages allow programmers to write a stencil specification in a high-level syntax, which is architecture independent.

In PATUS the user writes the stencil code in a C-like syntax: the tool then generates C code, by mean of a Strategy (i.e., optimization and parallelization methods such as cache-blocking). An auto-tuning phase is executed after the code generation: it is used to select an optimal or near optimal parameter configuration for the chosen stencil kernel, Strategy, and hardware platform.

Chapter 8

Performance Evaluation

The purpose of benchmarking and performance evaluation is to assess the performance and understand the characteristics of HPC platforms. A state-of-the-art solution to determine the performance of a system is to use LINPACK [37], to solve a dense system of linear algebraic equations, using LU factorization with partial pivoting. Since the problem is very regular, the performance achieved is quite high, and the performance numbers give a good correction of peak performance.

The most cited supercomputing ranking, the Top500 [9], lists the 500 most powerful commercially available computer systems and is based on HPL (High Performance LINPACK). Another way of assessing the performance could be by solving a real-life problem using a commercial software/code. The primary goal of such practice is the search for the best suitable machines for industrial/research projects. The downside of such methodology is that it usually only investigates one dimension of the overall performance, i.e., the results of a specific benchmark.

8.1 Performance Analysis

HPC applications seldom exploit the full potential of the modern supercomputers, due to the complexity of their architectures, both in terms of hardware and software, as described in Chapter 2 and Chapter 3. The emerging many-core and multi-socket nodes introduce additional degrees of complexity to the already high dimensionality and complexity of performance optimization. In Figure 8.1 is shown the cycle of per-

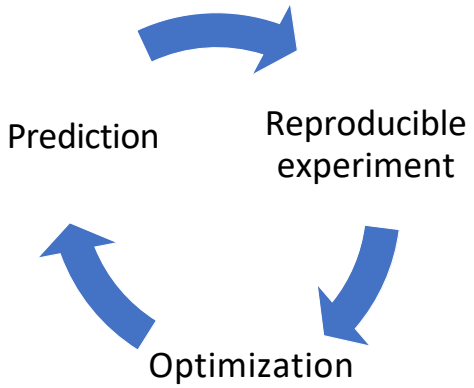

Figure 8.1: *The performance engineering cycle consists of three phases: analisys and prediction of the code's characteristics, reproducible experimentation and measurements of its run-time behavior, and optmization*

formance engineering: due to the trends in computer architectural complexity of the new computer systems and the complexity of the software that runs on them, the gap between expected and realized performance is widening.

Performance engineering is critical to address such a gap, with its cycle of prediction, reproducible measurement, and optimization. At the time of writing, the measurement is seldom referenced as Reproducible, and with this work, such an issue is being addressed.

Performance optimization not only requires to identify which are the code fragments that act as bottlenecks but also to understand the causes of said bottlenecks and determine a possible solution to improve the performance of the code under evaluation. While the identification of code segments that represent a bottleneck can be easily achieved by using timers, the characterization of the cause of the bottleneck requires more sophisticated analysis and modeling of the performance. The execution of a segment of code can no longer be easily traced due to some optimizations and features of the new micro-architectures: static or dynamic branch prediction, software prefetching, out-of-order execution.

8.1.1 Performance Models

For a long time, performance modeling has been done analytically as shown in Hockney and Curington [67], which became famous with the

Table 8.1: *Timeline of performance modeling research and the most relevant publications.*

1993	Hockney and Curington [67].
1994	Boydl [19].
1999	Roofline Model [135].
2000	Nudd [97].
2000	Hoisie [68].
2001	Kerbyson [79].
2005	Kerbyson [80].
2006	Asanovic [11].
2008	Suleman [128].
2010	ECM Model [131].

name of Roofline Model [135] in 2009. The assumption that drove Williams to explore and improve the model proposed by Hockney and Curington is that while stochastic analytical models [11] and statistical performance models [19] can predict performance, they seldom provide an actual insight of how to optimize the code or the usage of the compilers. The Roofline Model provides a sketch of the behavior of a specific architecture combining arithmetic intensity, memory performance (bandwidth) and computing performance (floating point operations).

The Execution-Cache-Memory [61, 131, 122](ECM) model refines the performance modeling of streaming loop kernels, and allows, in case of strong bandwidth limitations, a reasonably accurate prediction of the saturation point. Previously, it could only be addressed in a phenomenological way [128]. There are other examples of performance modeling not restricted to the single node analysis, describing an effort in the direction of large-scale modeling [68, 97, 79, 80]. At the basis of all the performance analysis, there is the execution of some code, which happens on a single chip. For this reason, the focus this work addresses the single node performance analysis.

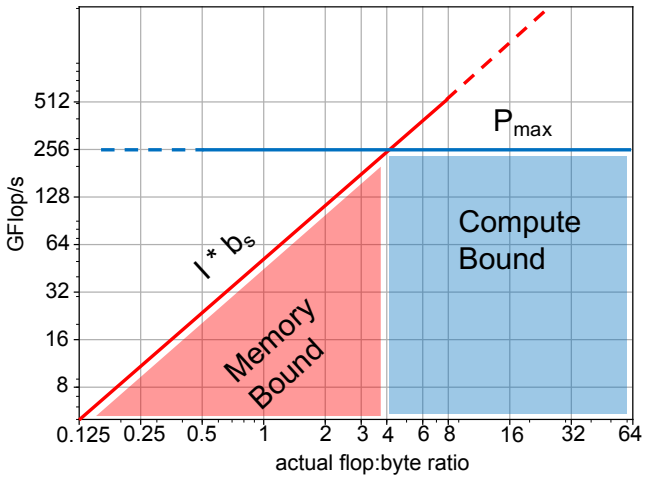

Figure 8.2: *Visualization of the Roofline model with explicit depiction of the compute bound area, delimited by the peak floating point performance P_{max}, and memory bound area, delimited by the applicable peak bandwidth b_s at a given arithmetic intensity I.*

Roofline Model

The Roofline model, described in Section 6.4, allows to understand the performance limit of an application, based on operational intensity (algorithm specific) and on memory bandwidth (hardware specific). The formula to compute this model is:

$$P = min(P_{max}, I * b_s),$$

where P_{max} is the peak floating point performance of a loop (assuming that data comes from L1 cache), b_s is the applicable peak bandwidth of the slowest data path utilized, and I is the arithmetic intensity ("work" per Byte transferred) over the slowest data path utilized ("the bottleneck").

The Roofline model yields an absolute upper performance bound for a loop, as shown in Figure 8.2. It assumes that the runtime is dictated either by the computational work or the data transfers to and from a single level in the memory hierarchy. Such an assumption entails that all data transfers across the whole memory hierarchy perfectly overlap with each other and with the execution of instructions in the core, which is too optimistic in the general case [64].

Stencil kernels usually have low arithmetic intensity, which means that on the roofline graph their upper bound is the memory bandwidth.

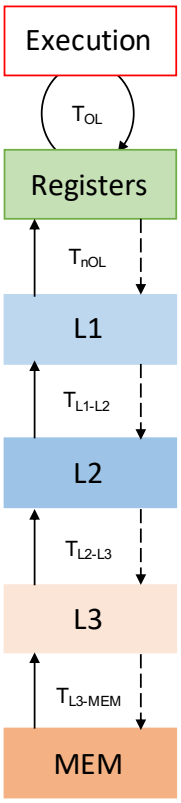

Figure 8.3: *Visualization of the factors involved in the calculation of the ECM model: overlapping time of computations and store (T_{OL}), time for loading the data from L1 to the registers (T_{nOL}), time for loading the data from L2 to L1 (T_{L1-L2}), time for loading from L3 to L2 (T_{L2-L3}), and time for loading from memory to L3 (T_{L3-MEM}).*

The Roofline model must, in principle, be generated only once for a given architecture and then can be reused and add to it the value of the kernels when tested. Some ceilings can be added to the Roofline and used to guide the optimizations to perform, as explained in Williams et al. [135].

ECM Model

Another state of the art model that one may use to analyze an architecture is the Execution Cache Memory model [61, 122]. It shares the idea of the Roofline that either the execution of the instructions or the data transfer time represent the limitation of the loop, thus driving its execution time. The difference lays in the fact that according to the ECM model, all the memory levels contribute to a single bottleneck: sometimes the transfer time can overlap (as in the Roofline), while other times it adds up. To pro-

vide a prediction of the performance, it takes into account the volume of the data moved to and from each level of cache, involving the inter-cache throughputs, and the actual computation that happens into the core (see Figure 8.3). The formula used to compute the prediction is the following:

$$T_{ECM,Mem} = max(T_{OL}, T_{nOL} + T_{L1-L2} + T_{L2-L3} + T_{L3-MEM},$$

where T_{OL} represents the overlapping time of computations and store, T_{nOL} is the time for loading the data from L1 to the registers, T_{L1-L2} the time for loading from L2 to L1 and so on.

Both Roofline and ECM model are based on a bottleneck analysis under a throughput assumption, but in the latter, the contributions of all memory hierarchy levels sum up to a single bottleneck: data transfers to different memory levels may overlap (like in the Roofline model) or may add up.

Whichever is the model used to analyze a specific stencil on a given architecture, such a process will drive to performance laws and will show the path to improve the implementations of the original problem. On the other side, the measured performance can also drive to adaptation and improvements in the performance models.

8.1.2 Grey Box Modeling

Performance models can either be analytically calculated or applied with the help of grey box tools, that help in gathering the information required to apply said models. The following subsections describe some of such tools.

ExaSAT

Exascale Static Analysis Tool [132] statically analyzes an application and automatically gather key characteristics about the computation, communication, data access patterns and data locality that are important in characterizing the performance of combustion codes. Loop-level information is collected to create a profile of the code, that is then analyzed to obtain a dependency graph and performance modeling. The performance model is parameterized with machine specifications, employing an abstract, simplified machine model, user (input) parameters (e.g., problem size), and software optimizations (e.g., loop fusion). The compiler analysis for ExaSAT is built on top of the ROSE compiler framework [105], an

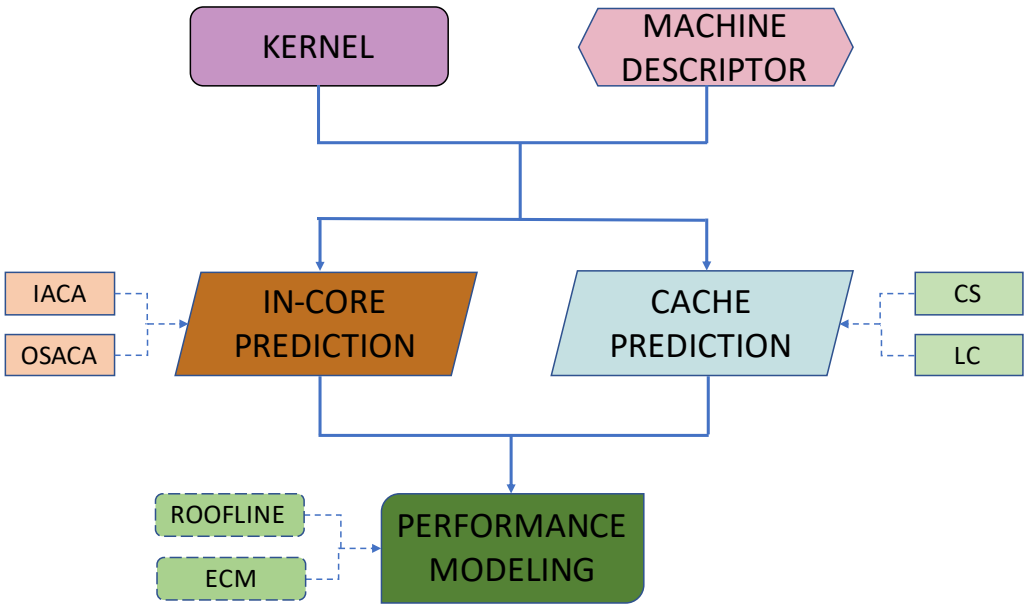

Figure 8.4: *Overview of Kerncraft: the user provides kernel code, constants, and a machine descriptor. IACA or OSACA, pycachesim, and a compiler are employed to build the ECM and Roofline models. Figure adapted from*

open-source compiler infrastructure developed at Lawrence Livermore National Laboratory. ExaSAT is restricted to the Roofline model for performance prediction.

Empirical Roofline Toolkit

The Roofline model provides an intuitive model and a graphic visualization for understanding kernel performance on various architectures, but suffers from three factors: hardware characterization (the construction of the roofline model requires expert knowledge of the target processor microarchitecture), knowledge of the characteristics of the kernel, and software characterization (requires an expert in the architecture and the algorithm to predict the performance of the HW/SW stack when compiling/running a routine). The Empirical Roofline Toolkit (ERT), firstly introduced in [85], empirically determines the machine characteristics (CPU or GPU-accelerated) that are needed to generate the Roofline model. An example of the required information are the maximum bandwidth for the various levels of the memory hierarchy and the maximum GFlop/s rate, obtained using several "micro-kernels".

Kerncraft

Kerncraft is a tool, implemented at the FAU Erlangen-Nürnberg, to analyze a C like source code containing a loop nest (stencil kernel loop) in terms of data transfers and code to be executed. The user must also provide an architecture description file (in YAML format) and select what performance model to apply.

A schematization of the full pipeline of Kerncraft is shown in Figure 8.4. A complete description of the machine descriptor and the pseudo-C code constraint is showed in [64].

The prediction of the caching behavior is performed through two possible models: Layer Condition [122, 63] or Cache Simulation. Once evaluated the contribution of the data movements, it is necessary to asses the in-core execution behavior. It is done using the Intel Architecture Core Analyzer [7] (IACA), that, using as input assembly code representing a series of instructions, predicts throughput and latency. Its limitation is that it only supports Intel® 64 code, including Intel® AVX, AVX2, and AVX-512 instructions. OSACA [8], an open source alternative to IACA, is under development, but still not ready to be fully integrated into Kerncraft.

Kerncraft, differently than ExaSAT and ERT, supports automatic detection of the topology of the architecture where the analysis is conducted, and does not rely on compiler-generated loops, which usually introduce an element of uncertainty. Additionally, thanks to the use of IACA (and in the next future OSACA), it generates more accurate in-core predictions than the other competitors, that use a simplified machine model. For these reasons, Kerncraft has been selected as the tool to model the performance in the experiments that complement this work.

8.2 Performance Measurement

The results of the experiments that are part of this thesis are an average performance over ten executions. Additionally, a standard deviation is represented on the graphs. It can be argued that a median of the performance could be more appropriate, and that is probably true in other domains. The idea of the author is that, in the HPC domain, each of the kernels considered in this work, would be normally executed thousands, or even millions, of times, so that the average represents an appropriate metric. The quantities that need to be measured are the execution time

and the number of floating point operations (FLOP) per run. While the execution time is measured by using the *gettimeofday* routine available in the timing library, the number of FLOPs is automatically calculated by inspecting a kernel. GFLOP/s represents billions (10^9) of floating-point operations per second.

By analyzing a code, it is possible to discover if its demands exceed the architecture's capabilities, thus affecting the performance. In the stencil kernels, the computational work can be identified with multiplications, additions and sometimes divides. Consequently, the performance is assessed in GFLOP/s.

Another possible metric is lattice updates per second (LUP/s), which scales the GFLOP/s by the number of FLOPs executed. The independence of this metric from the number of FLOPs executed represent both its strength and weakness: it emphasizes the work done decoupling it from possible optimizations that interfere with the arithmetic intensity, since such optimizations may change the number of FLOPs per stencil update. For said reason, the results presented use GFLOP/s as a metric, even though during the experiments also the LUP/s are computed and stored for further analyses.

To assess the data movements, i.e., the communication between the CPU and the memory, the work is defined as the number of bytes transferred, including the traffic due to cache misses and speculative transfers (e.g., prefetching). The bandwidth is measured in bytes transferred per second (B/s or millions of bytes per second GB/s).

The ratio between memory bandwidth and peak performance (measured in FLOP/s) is called Machine Balance (B_m) [22, 62]. A complementary concept is the code balance (B_c) of a loop, defined as the ratio between the data traffic and the number of FLOPs. Its reciprocal is called arithmetic intensity, or computational intensity, e.g., the ratio between the computation and the communication of a given kernel. The expected maximum fraction of the peak performance that a code with a balance B_c can reach on a machine with balance B_m is given by:

$$min(1, \frac{B_m}{B_c})$$

8.2.1 Cache Misses

Cache misses can be classified into three basic types [66]: compulsory, capacity, or conflict misses.

Compulsory misses, also called cold start misses, happen when the first access to a block is not in the cache, so it must be loaded for the first time.

If the cache cannot contain all the blocks needed during the execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.

If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called collision misses or interference misses.

8.2.2 Code Structure and Parallelism

Structured grid codes often perform a large number of sweeps over a grid applying a stencil operator, i.e., a function that relates a point in the grid to its neighborhood. Sweeps represent the evolution over time of a problem, in codes like parabolic or hyperbolic PDEs. In elliptic PDEs, they are used for the convergence, while in multigrid codes they modify the resolution of the grid. Inherent parallelism can vary widely from one sweep to another: Gauss-Seidel method, upwinding stencils, Red-Black Gauss-Seidel method, Jacobi's method. This work is restricted to the Jacobi's iteration, described in [62].

8.2.3 Memory Access Pattern and Data Locality

The memory access pattern of a stencil can have dramatic impacts on the performance. In this section, we examine the memory access pattern and cache locality for a 2D problem on rectangular Cartesian grids and allow the reader to contemplate more complicated problems. In Figure 8.5 is shown the access pattern of a 2D 5-point stencil using a Jacobi algorithm. The C language implements a row-major order for multidimensional arrays: the inner loop variable must ensure stride-one access in order to minimize the memory traffic. Figure 8.5 shows how the row-wise traversal takes place: the stencil point with the biggest j coordinate (leading point) is brought to cache for the first time, causing a compulsory cache miss. This value will stay in cache for three row traversals if the cache of the machine in use is big enough to store more than two rows of the grid, so the number of required loads is not five but four (plus one store). In

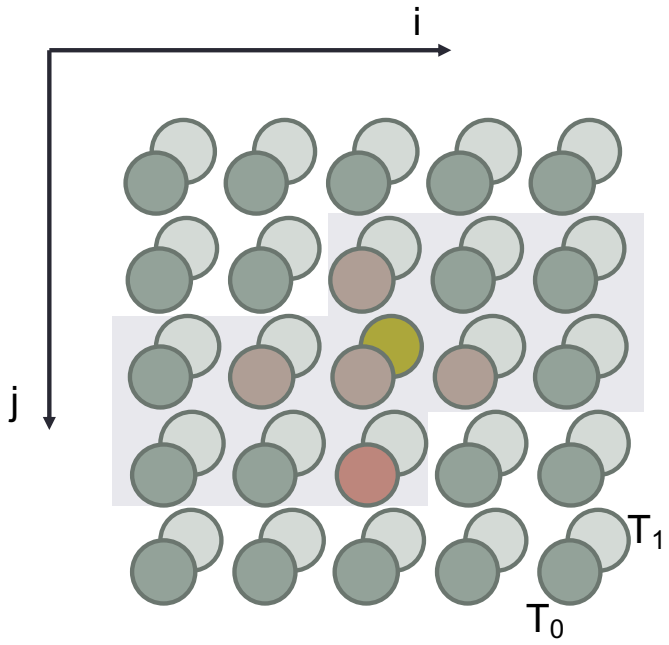

Figure 8.5: *Jacobi algorithm for a 2D 5-point stencil update: in pink and red the points needed at time T_0 to obtain the point in yellow at time T_1. The shadow indicates the points that are needed for the actual computation: if at least two successive rows can be kept in the cache, only one cell per update has to be fetched from memory (in red).*

three dimensions, the distance between the leading and trailing points in the stencil should be twice the plane size, thus representing a challenge to cache capacities. Such an analysis,called Layer Condition analysis, is automatically performed by Kerncraft.

An Intel Xeon E5-2640 v4 with a clock of 2.6 GHz has maximum memory bandwidth (b_{max}) of 51.6 GB/s (measured via STREAM benchmark). We can thus compute:

$$P_{max} = 10 cores * 2.6 GHz * 16 DPFLOP = 416 GFLOP/s.$$

$$B_m = \frac{b_{max}}{P_{max}} = \frac{51.6 GB/s}{416 GFLOP/s} = 0.124 \frac{B}{FLOP} = 0.016 \frac{W}{FLOP}.$$

If the cache is big enough to hold two successive rows, when executing a 2D 5-point Jacobi stencil, loading the neighbours at j and j-1 are already in cache, so only 1 load and 1 store plus the write allocate take place, so

$$B_c = data\ traffic/number\ of; FLOPs = 3W/5FLOP = 0.6W/FLOP.$$

The expected performance in GFLOP/s is then:

$$P = min(P_{max}, P_{max} * B_m/B_c) = 10.74 GFLOP/s.$$

8.2.4 Optimization of Stencil Codes

The stencil problem has been widely studied due to its importance and occurrence in computational sciences. Several optimization techniques have been proposed over time.

Stencil codes are often easy to parallelize, but it is difficult to obtain a good ratio of the peak performance, due to their usually low arithmetic intensity. The techniques proposed, either act on the loop structure, rearranging it, but performing exactly the same operations, or on the algorithm, dramatically changing the number of operations required.

To the first group belong cache blocking and time skewing, that are implementation-only optimizations, where the loops are restructured to improve performance. To the second group of optimizations belong multigrid and adaptive mesh refinement.

Since it is complicated to rearrange the loops, ad hoc stencil compilers have been developed and studied: between these, we find Pochoir, Halide, PLUTO, and PATUS.

Recently there are efforts to port stencil codes to GPUs via compilers like TOAST [111].

In this work, we consider two stencil compilers that use different approaches: PLUTO [18], a source to source C compiler that exploits the polyhedral model to find affine transformations for efficient tiling both in space and time, and PATUS [26] that uses a DSL language to effectively and productively describe a stencil and then auto-tunes the generated code according to a predefined strategy.

Cache Blocking

As previously discussed, the capacity of the cache affects the number of cache misses. If we consider the example of stencil sweeps in 8.2.3, it is possible to block the loops in a way that allows holding in the cache a useful working set, a well-known concept deriving from the cache blocking techniques applied to dense matrix-matrix multiplication.

In two dimensions is necessary to block only the unit-stride loop, while in three dimensions either the unit-stride, the middle dimension,

or both loops need to be blocked to maintain a cache-friendly working set.

Time Skewing

A stencil is applied multiple times, over several sweeps. Cache blocking only works inside a single sweep. The same concept of blocking the loops could also be applied to the time loop, thus obtaining a space-time blocking. In this way, once the points of interest are in cache, they advance in time, to maximize their reuse, and possibly drive to an increase of the arithmetic intensity. It must be noted that such a technique is limited both by the bandwidth to the cache and the in-core performance.

Cache-oblivious algorithms were applied to structured grid codes [44, 78] organizing the space-time in trapezoids and parallelepipeds, which are traversed in a recursive ordering. Such a way of traversing is so expensive to cancel the benefits given by the reduction in cache misses, thus resulting in even slower code. Cache aware implementations have been introduced [91, 119, 121, 137]: they adopt the idea of dividing the space-time into trapezoids and parallelepipeds but use complex loop nests instead of the recursion. The code complexity drastically increases with the number of dimensions that are blocked.

Chapter 9

Experimental Testbeds

We set up and execute two macro-experiments (as defined in Section 4.1) using up to three different methods, on two systems. A precise description of the experiment, in terms of *(Problem, Method, System)*, is a base step towards the reproducibility of the research.

As sustained in Sections 5.1, sharing the source code is beneficial, but its availability is not sufficient for reproducibility. In fact the code may not compile, or the results could be affected by the differences of other components in the software stack. Pieces of information such as version of the compiler, compilation flags, configurations, experiment parameters, and raw results are fundamental for the reproducibility of an experiment.

The most important conference in the field of high performance computing, SuperComputing, has since its 2016 edition, launched a reproducibility effort, inviting the authors to submit, together with their papers, an artifact description, i.e. an appendix describing the details of their software environments and computational experiments, so that an independent person could replicate their results. Such an appendix will be mandatory for papers submitted into the main track, starting from SuperComputing 2019.

It is worth noting that the information stored by PROVA! provides all the necessary fields to fill such an appendix, plus additional details that may be used for further analysis, a posteriori. Configurations, methods, source code, *methodTypes* (with their respective run instructions and *easyconfigs*) used in the experiments later discussed in this work, are available

at [54].

9.1 Systems

The experiments that have been performed in this work, unless differently specified, have been executed in two high performance computing facilities, geographically far away from each other, composed of compute nodes with different architectures. The stencil experiments have been run on a single node of the Emmy and miniHPC clusters.

The software stack is maintained by PROVA! v0.3 (on both clusters) and, at each time, only the needed software is present in PATH.

9.1.1 Validation Macro-Experiment

For our tests the module "OpenMP : GCC/7.3.0-2.30", installed through EasyBuild [70] [48] was loaded. The OpenMP module is used to test the prediction offered by Kerncraft (v0.6.10, using GCC/7.3.0-2.30). STEMPEL, v0.1.0, has been used to generate the stencil kernels and automate the execution of the experiments by calling both Kerncraft and PROVA!. A description of said process is detailed in Appendix C.

Likwid [112] is present as system dependency.

9.1.2 Performance Engineering Cycle Macro-Experiment

During this macro-experiment, the following modules, installed through EasyBuild [70] [48] were loaded:

- OpenMP : GCC/7.3.0-2.30
- PATUS : GCC/7.9.0-2.30, PATUS/0.1.4, Java/1.7.0_79, Maxima/5.37.2 (compiled with ecl/16.0.0)
- PLUTO-pet : GCC/7.3.0-2.30, PLUTO-pet/0.11.0 (pet branch)

Likwid [112] is present as system dependency.

After the validation phase, we pass to the optimization and comparison analysis: we use PROVA! to first repeat an experiment using one compiler only. Then we can re-experiment, using another method, in this case, another compiler, to generate code for the chosen stencil. Subsequently, we move to another system and use the same code as before, thus having a repetition and a re-experimentation locally, and globally a

replication. Each repetition gives us an insight about the compiler performance on a chosen system. Re-experimentation allows us to compare different compilers since they run on the same machine. Porting the experiment to another system enables studying the behaviour of the single compilers on another architecture.

9.1.3 Emmy

The Emmy cluster (NEC), located at the Regionales RechenZentrum Erlangen (RRZE), is a high-performance compute resource with high speed interconnect. It is intended for distributed-memory (MPI) or hybrid parallel programs with medium to high communication requirements. It is composed by:

- 560 compute nodes, each with two Xeon 2660v2 "Ivy Bridge" chips (10 cores per chip + SMT) running at 2.2 GHz with 25 MB Shared Cache per chip and 64 GB of RAM
- 2 frontend nodes with the same CPUs as the nodes
- 16 Intel Xeon Phi coprocessors
- 16 Nvidia K20 GPGPUs spread over 16 compute nodes
- parallel filesystem (LXFS) with a capacity of 400 TB and an aggregated parallel I/O bandwidth of more than 7000 MB/s
- fat-tree Infiniband interconnect fabric with 40 GBit/s bandwith per link and direction
- overall peak performance of ca. 234 TFlop/s
- 191 TFlop/s LINPACK, using only the CPUs
- Peak performance $P_{max} = 176\ GFLOP/s$
- Memory bandwidth $b_{max} = 40.05\ GB/s$ per socket, measured with STREAM benchmark (through *likwid-bench*)
- Machine balance $B_m = 0.227\ B/FLOP$

A representation of its topology, retrieved via Likwid, is shown in Listrings 9.1.

9.1.4 MiniHPC

MiniHPC is a high-performance cluster with high speed interconnect, located at the University of Basel, whose details are:

- 22 compute nodes, each with two Xeon E5-2640 v4 "Broadwell" chips (10 cores per chip + SMT) with 25 MB Shared Cache per chip and 64 GB of RAM

- the compute nodes run at a frequency of 2.6 GHz, using the performance governor provided by the Intel Pstate driver
- 1 frontend node with the same CPUs as the nodes
- 4 Intel Xeon Phi 7210 (64 cores per chip + SMT) running at 1.3 GHz with 32 MB Shared Cache per chip and 96 GB of RAM
- fat-tree Intel Omni-Path interconnect fabric with 100 GBit/s bandwith
- overall peak performance of ca. 23.6 TFlop/s
- 18.3 TFlop/s peak performance, using only the CPUs
- Peak performance $P_{max} = 416\ GFLOP/s$
- Memory bandwidth $b_{max} = 51.60\ GB/s$ per socket, measured with STREAM benchmark (through *likwid-bench*)
- Machine balance $B_m = 0.124\ B/FLOP$

A representation of its topology, retrieved via Likwid, is shown in Listings 9.2.

```
CPU name: Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz
CPU type: Intel Xeon IvyBridge EN/EP/EX processor
CPU stepping: 4
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:    2
Cores per socket: 10
Threads per core: 2
--------------------------------------------------------------------------------
HWThread  Thread    Core    Socket    Available
0    0    0    0    *
1    0    1    0    *
2    0    2    0    *
3    0    3    0    *
4    0    4    0    *
5    0    8    0    *
6    0    9    0    *
7    0    10    0    *
8    0    11    0    *
9    0    12    0    *
10    0    0    1    *
11    0    1    1    *
12    0    2    1    *
13    0    3    1    *
14    0    4    1    *
15    0    8    1    *
16    0    9    1    *
17    0    10    1    *
18    0    11    1    *
19    0    12    1    *
20    1    0    0    *
21    1    1    0    *
22    1    2    0    *
23    1    3    0    *
24    1    4    0    *
25    1    8    0    *
26    1    9    0    *
27    1    10    0    *
28    1    11    0    *
29    1    12    0    *
30    1    0    1    *
31    1    1    1    *
32    1    2    1    *
33    1    3    1    *
34    1    4    1    *
35    1    8    1    *
36    1    9    1    *
37    1    10    1    *
38    1    11    1    *
39    1    12    1    *
--------------------------------------------------------------------------------
Socket 0:    ( 0 20 1 21 2 22 3 23 4 24 5 25 6 26 7 27 8 28 9 29 )
Socket 1:    ( 10 30 11 31 12 32 13 33 14 34 15 35 16 36 17 37 18 38 19 39 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:      1
Size:     32 kB
Cache groups:    ( 0 20 ) ( 1 21 ) ( 2 22 ) ( 3 23 ) ( 4 24 ) ( 5 25 ) ( 6 26 )
( 7 27 ) ( 8 28 ) ( 9 29 ) ( 10 30 ) ( 11 31 ) ( 12 32 ) ( 13 33 ) ( 14 34 ) ( 15 35 )
( 16 36 ) ( 17 37 ) ( 18 38 ) ( 19 39 )
--------------------------------------------------------------------------------
Level:      2
Size:     256 kB
Cache groups:    ( 0 20 ) ( 1 21 ) ( 2 22 ) ( 3 23 ) ( 4 24 ) ( 5 25 ) ( 6 26 )
( 7 27 ) ( 8 28 ) ( 9 29 ) ( 10 30 ) ( 11 31 ) ( 12 32 ) ( 13 33 ) ( 14 34 ) ( 15 35 )
( 16 36 ) ( 17 37 ) ( 18 38 ) ( 19 39 )
--------------------------------------------------------------------------------
Level:      3
Size:     25 MB
Cache groups:    ( 0 20 1 21 2 22 3 23 4 24 5 25 6 26 7 27 8 28 9 29 )
( 10 30 11 31 12 32 13 33 14 34 15 35 16 36 17 37 18 38 19 39 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:    2
--------------------------------------------------------------------------------
Domain:     0
Processors:    ( 0 20 1 21 2 22 3 23 4 24 5 25 6 26 7 27 8 28 9 29 )
Distances:    10 21
Free memory:    11728 MB
Total memory:    32734.4 MB
--------------------------------------------------------------------------------
Domain:     1
Processors:    ( 10 30 11 31 12 32 13 33 14 34 15 35 16 36 17 37 18 38 19 39 )
Distances:    21 10
Free memory:    28665.2 MB
Total memory:    32768 MB
--------------------------------------------------------------------------------
```

Listing 9.1: *Topology of a node of Emmy, obtained via likwid-topology*

```
CPU name: Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
CPU type: Intel Xeon Broadwell EN/EP/EX processor
CPU stepping: 1
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:    2
Cores per socket: 10
Threads per core: 1
--------------------------------------------------------------------------------
HWThread  Thread    Core    Socket    Available
0    0    0    0    *
1    0    1    0    *
2    0    2    0    *
3    0    3    0    *
4    0    4    0    *
5    0    5    0    *
6    0    6    0    *
7    0    7    0    *
8    0    8    0    *
9    0    9    0    *
10    0    10    1    *
11    0    11    1    *
12    0    12    1    *
13    0    13    1    *
14    0    14    1    *
15    0    15    1    *
16    0    16    1    *
17    0    17    1    *
18    0    18    1    *
19    0    19    1    *
--------------------------------------------------------------------------------
Socket 0:    ( 0 1 2 3 4 5 6 7 8 9 )
Socket 1:    ( 10 11 12 13 14 15 16 17 18 19 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:      1
Size:     32 kB
Cache groups:    ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ( 6 ) ( 7 ) ( 8 ) ( 9 )
( 10 ) ( 11 ) ( 12 ) ( 13 ) ( 14 ) ( 15 ) ( 16 ) ( 17 ) ( 18 ) ( 19 )
--------------------------------------------------------------------------------
Level:      2
Size:     256 kB
Cache groups:    ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ( 6 ) ( 7 ) ( 8 ) ( 9 )
( 10 ) ( 11 ) ( 12 ) ( 13 ) ( 14 ) ( 15 ) ( 16 ) ( 17 ) ( 18 ) ( 19 )
--------------------------------------------------------------------------------
Level:      3
Size:     25 MB
Cache groups:    ( 0 1 2 3 4 5 6 7 8 9 ) ( 10 11 12 13 14 15 16 17 18 19 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:    2
--------------------------------------------------------------------------------
Domain:     0
Processors:    ( 0 1 2 3 4 5 6 7 8 9 )
Distances:    10 21
Free memory:    263.93 MB
Total memory:    32671.7 MB
--------------------------------------------------------------------------------
Domain:     1
Processors:    ( 10 11 12 13 14 15 16 17 18 19 )
Distances:    21 10
Free memory:    27457.6 MB
Total memory:    32768 MB
--------------------------------------------------------------------------------
```

Listing 9.2: *Topology of a node of MiniHPC, obtained using the utility likwid-topology*

Table 9.1: *Table representing all the combinations of the characteristic parameters of a star stencil, depicting the values used for the validation experiments.*

Stencil Pattern	**Cardinality**	**Coefficients**	**Symmetricity**	**Radius**	**Grid size**
STAR	2D	Homogeneous	Constant	1-4	12000 x 12000
		Heterogeneous			
		Point-symmetric	Variable	1-4	5000 x 5000
		Isotropic			
	3D	Homogeneous	Constant	1-4	250 x 250 x 250
			Variable	1-4	
		Heterogeneous	Constant	1-4	500 x 500 x 500
		Point-symmetric			
		Isotropic	Variable		250 x 250 x 250

Table 9.2: *Table representing all the combinations of the characteristic parameters of a box stencil, depicting the values used for the validation experiments.*

Stencil Pattern	**Cardinality**	**Coefficients**	**Symmetricity**	**Radius**	**Grid size**
BOX	2D	Homogeneous	Constant	1-4	3000 x 3000
		Heterogeneous			
		Point-symmetric	Variable	1-3	
		Isotropic			
	3D	Homogeneous	Constant	1-2	500 x 500 x 500
		Heterogeneous			
		Point-symmetric	Variable		250 x 250 x 250
		Isotropic	Constant	1-3	500 x 500 x 500
			Variable	1-2	250 x 250 x 250

9.2 Problems

Using STEMPEL, it is relatively easy to generate synthetic stencil kernels, that cover all the characteristic parameters identified and described in Section 7.2.

9.2.1 Kernels Used in the Validation Macro-Experiment

The problems that have been analyzed, implemented, and executed cover a vast majority of the possible combinations of dimensionality, symmetricity, type of stencil, type of the coefficients, as schematized in TABLE 9.1 and TABLE 9.2.

The codes for the stencil kernels have been generated via STempEL (v0.1.0) and analyzed using Kerncraft (v0.6.10). For each of the kernels both ECM and Roofline, models have been applied. The sizes of input and output grids have generally been chosen so that they do not fit in L3, and the computation is carried out over grid points in IEEE double precision arithmetic. The grid size may have been occasionally scaled down, mostly for box stencils and when using variable coefficients, to allow Kerncraft and IACA to perform the analysis of the kernel. The number of timesteps to execute has been automatically and dynamically selected by making sure that the overall execution time of each kernel lasts at least 2 seconds. The threads used for the measurement are 1 and 10 (full socket), explicitly pinning to the logical nodes, as described in Section 6.3.

```
1:  double a[M][N], b[M][N];
2:  double c0, c1, c2;
3:
4:  for(int j=1; j < M-1; j++){
5:    for(int i=1; i < N-1; i++){
6:      b[j][i] = c0 * a[j][i]
7:          + c1 * (a[j][i-1] + a[j-1][i] +
8:            a[j+1][i] + a[j][i+1])
9:      + c2 * (a[j-1][i-1] + a[j+1][i-1] +
10:       a[j-1][i+1] + a[j+1][i+1]);
11:   }
12: }
```

Listing 9.3: *Kernel of a 2-dimensional radius 1 isotropic box stencil with constant coefficients*

9.2.2 Kernels Used in the Performance Engineering Cycle

A stencil has been selected to be used for a second experiment, with the goal of showing a complete performance engineering cycle (discussed in Chapter 8).

In this case has been executed a 2-dimensional, box stencil with isotropic and constant coefficients. Two different values for the radius have been used, in order to explore and analyze the behavior of the compilers, the performance models and the architectures when varying the neighbors involved and thus the operations to be executed: radius 1 (yielding 11 FLOPs), in Listing 9.3 and radius 4 (yielding 89 FLOPs), in Listing 9.4. Additionally, the experiment has been conducted using two different grid sizes, the first 500^2 and 3000^2 points in IEEE double precision arithmetic: in this way we explore the behavior of performance models, architectures and (stencil) compilers when the grids fit and do not fit in the L3 caches. The time-steps executed are 200, and the threads used for the measurement are in the range [1, 20] (up the the full node), explicitly pinning either to the logical nodes or spreading the threads over the sockets, as described in Section 6.3.

```
1:  double a[M][N], b[M][N];
2:  double c0, c1, c2, c3, c4, c5, c6, c7, c8;
3:
4:  for(int j=4; j < M-4; j++) {
5:      for(int i=4; i < N-4; i++) {
6:      b[j][i] = c0 * a[j][i]
7:      + c1 * (a[j][i-1] + a[j-1][i] + a[j+1][i] + a[j][i+1])
8:      + c2 * (a[j][i-2] + a[j-1][i-1] + a[j+1][i-1] +
9:          a[j-2][i] + a[j+2][i] + a[j-1][i+1] +
10:         a[j+1][i+1] + a[j][i+2])
11:     + c3 * (a[j][i-3] + a[j-1][i-2] + a[j+1][i-2] +
12:         a[j-2][i-1] + a[j+2][i-1] + a[j-3][i] +
13:         a[j+3][i] + a[j-2][i+1] + a[j+2][i+1] +
14:         a[j-1][i+2] + a[j+1][i+2] + a[j][i+3])
15:     + c4 * (a[j][i-4] + a[j-1][i-3] + a[j+1][i-3] +
16:         a[j-2][i-2] + a[j+2][i-2] + a[j-3][i-1] +
17:         a[j+3][i-1] + a[j-4][i] + a[j+4][i] +
18:         a[j-3][i+1] + a[j+3][i+1] + a[j-2][i+2] +
19:         a[j+2][i+2] + a[j-1][i+3] + a[j+1][i+3] +
20:         a[j][i+4])
21:     + c5 * (a[j-1][i-4] + a[j+1][i-4] + a[j-2][i-3] +
22:         a[j+2][i-3] + a[j-3][i-2] + a[j+3][i-2] +
23:         a[j-4][i-1] + a[j+4][i-1] + a[j-4][i+1] +
```

```
24:         a[j+4][i+1] + a[j-3][i+2] + a[j+3][i+2] +
25:         a[j-2][i+3] + a[j+2][i+3] + a[j-1][i+4] +
26:         a[j+1][i+4])
27:     + c6 * (a[j-2][i-4] + a[j+2][i-4] + a[j-3][i-3] +
28:         a[j+3][i-3] + a[j-4][i-2] + a[j+4][i-2] +
29:         a[j-4][i+2] + a[j+4][i+2] + a[j-3][i+3] +
30:         a[j+3][i+3] + a[j-2][i+4] + a[j+2][i+4])
31:     + c7 * (a[j-3][i-4] + a[j+3][i-4] + a[j-4][i-3] +
32:         a[j+4][i-3] + a[j-4][i+3] + a[j+4][i+3] +
33:         a[j-3][i+4] + a[j+3][i+4])
34:     + c8 * (a[j-4][i-4] + a[j+4][i-4] + a[j-4][i+4] +
35:         a[j+4][i+4]);
36:     }
37: }
```

Listing 9.4: *Kernel of a 2-dimensional radius 4 isotropic box stencil with constant coefficients*

9.3 Methods

9.3.1 Methods Used in the Validation Macro-Experiment

The micro-experiments we set up, whose definition appears in Section 4.1, consist of a naive implementation of the problem, by parallelizing with OpenMP and using NUMA-aware initialization. The source code has been automatically generated by STEMPEL.

9.3.2 Methods Used in the Performance Engineering Cycle Macro-Experiment

The micro-experiments we set up, whose definition appears in Section 4.1, consist of the following triplet of methods:

- Naive implementation of the problem, by parallelizing with OpenMP and using NUMA-aware initialization
- Diamond tiling and loop transformation, according to the polyhedral model, by mean of PLUTO. It provides as output an OpenMP parallelized C code
- DSL + Auto-tuning approach using PATUS: from a stencil specification to C code which is then optimized automatically tuning the

hardware parameters specified, like cache block, chunk, unrolling factor

Chapter 10

Performance Benchmarking Experiments

The added value provided by PROVA! can be seen at the moment of running and analyzing the results of the experiments.

PROVA! allows to explore the dimensions of *Problem*, *Method*, and *System* that have been presented in Section 4.1, thus enabling *repetition*, *replication*, and *re-experimentation* (see Section 4.2).

10.1 Stencil Compilers

Merely assembling and comparing performance results obtained by the same compiler on different machines could have few or no meaning at all due to the architectural differences. Using PROVA! it is possible to compare them, evaluating the behavior they show on the different machines, thanks to the consistency of the environment. Ideally, users can choose the compiler offering the best performance on the system they have access to. Such an analysis gives value to the *replication*: focus on the behavior rather than on the numbers. Running an experiment on a machine allows evaluating how the performance changes while changing the parameters (i.e., dimension, number of threads used), helping to understand how a compiler behaves (e.g., if the performance scales to the whole).

OpenMP code is usually tuned for the target machine, thus making it senseless to compare two machines, based on the performance results

of such a code. When it comes to automatically generated code, on the other hand, it becomes essential to try and check if the generated code performs well on specific machines only, or if the compiler generates decently performing code for several machines.

The performance of each code must be evaluated in the light of the expected performance, obtained as a prediction when applying a performance model. Performance models could fit perfectly to a specific problem, thus resulting in an exact match between predicted and measured performance, or a mismatch could show up. In such cases the reason for the discrepancy should be investigated and addressed, either adapting the model or optimizing the code, to reflect the expected performance.

Using the code produced by several compilers as input, it is possible to characterize the architectures. Furthermore, it would be possible to identify and extrapolate the parameters that affect the execution of such codes.

Analyzing the outputs of *repetition* and *re-experimentation* experiments certainly gives information about the compilers and the validity of their solutions for the fixed system used. Going a little bit beyond, repeating said experiments on several systems and focusing not only (strictly) on the numerical outputs but also on the big picture, leads to what we call *behavioral insight*.

Remember Hamming's observation:

> "The purpose of computing is insight, not numbers."

10.1.1 Metric used for Compilers Evaluation and Comparison

Due to hardware differences, it is not possible (or valuable) to use the execution time as a metric: a possible solution to this problem is to compare against relative speed-up. It is not safe to assume that a compiler produces code resulting in the same speedup curves (both in terms of weak and strong scaling) on different machines. Another option is to use a metric of performance like Flop/s (Floating point operations per second) or LUP/s (Lattice Updates per second) and look at the way they scale over the number of threads used to characterize a code and an architecture.

Having an insight into the behavior of the scaling allows testing if it is bound to specific architectural characteristics of the machine or if

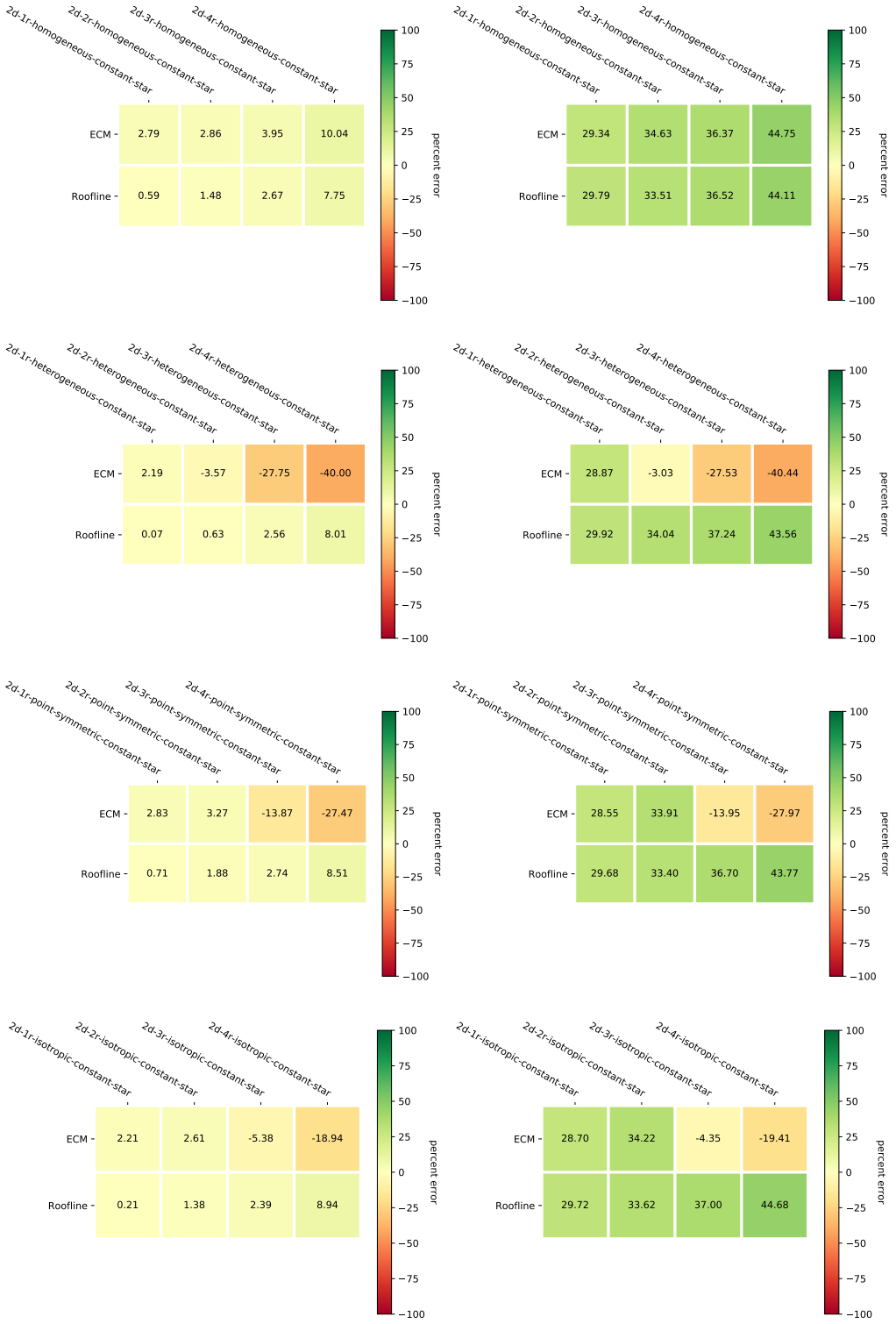

Figure 10.1: *Percentage error of the prediction obtained through ECM and Roofline model, applied to a 2-dimensional star stencil with homogeneous, heterogeneous, point-symmetric and isotropic constant coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through* PROVA!.

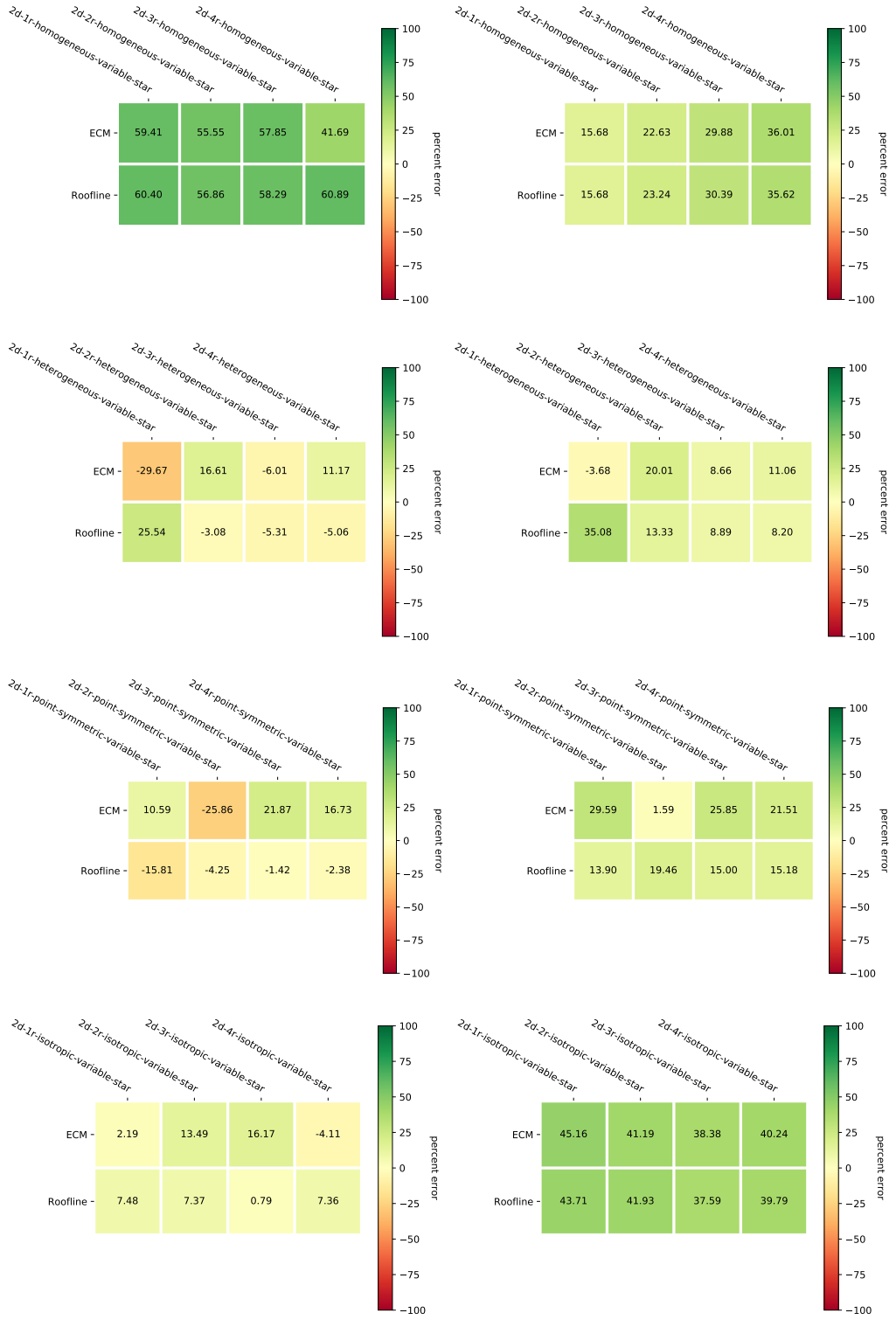

Figure 10.2: *Percentage error of the prediction obtained through ECM and Roofline model, applied to a 2-dimensional star stencil with homogeneous, heterogeneous, point-symmetric and isotropic variable coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through* PROVA!.

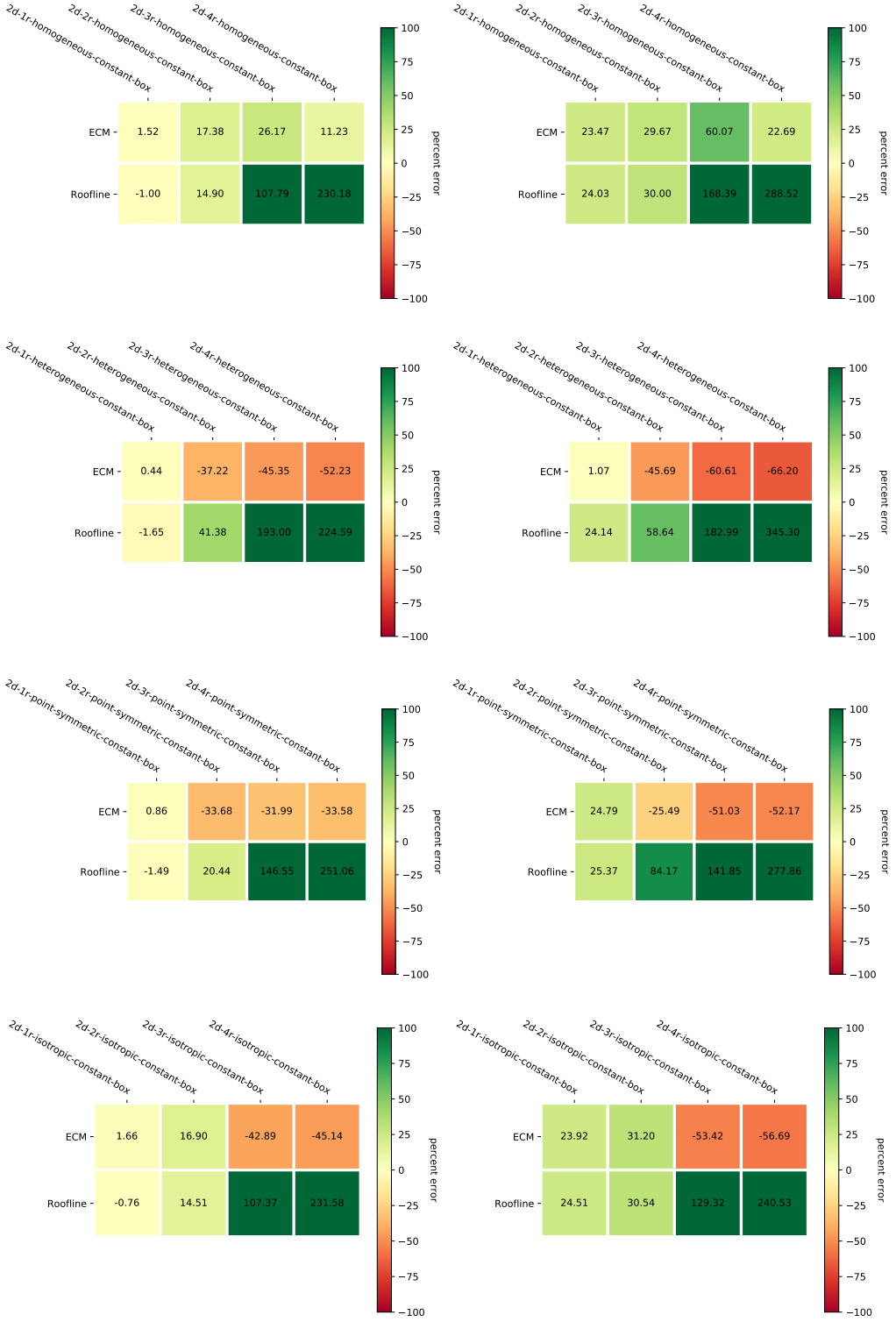

Figure 10.3: *Percentage error of the prediction obtained through ECM and Roofline model, applied to a 2-dimensional box stencil with homogeneous, heterogeneous, point-symmetric and isotropic constant coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through* PROVA!.

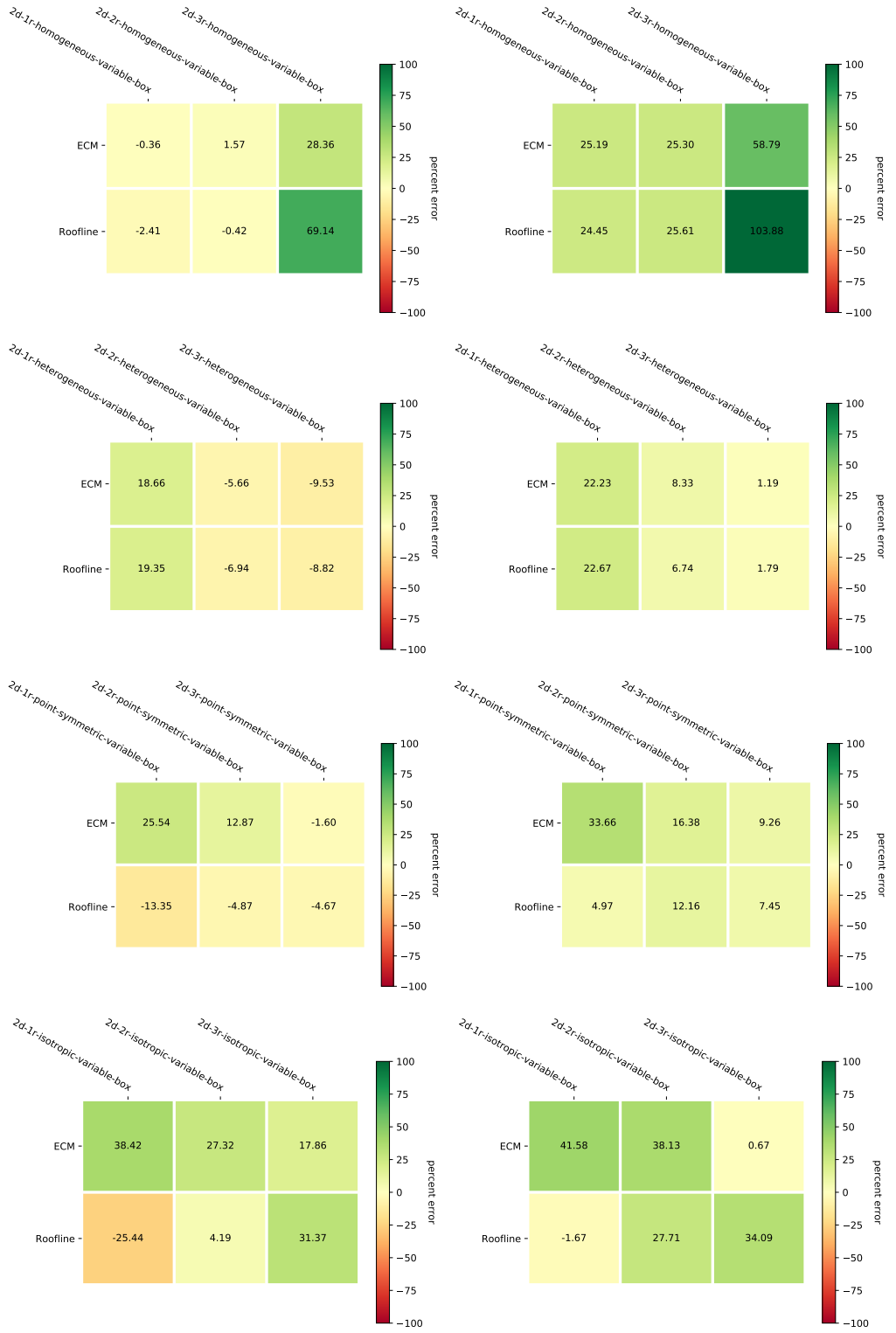

Figure 10.4: *Percentage error of the prediction obtained through ECM and Roofline model, applied to a 2-dimensional box stencil with homogeneous, heterogeneous, point-symmetric and isotropic variable coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through* PROVA!.

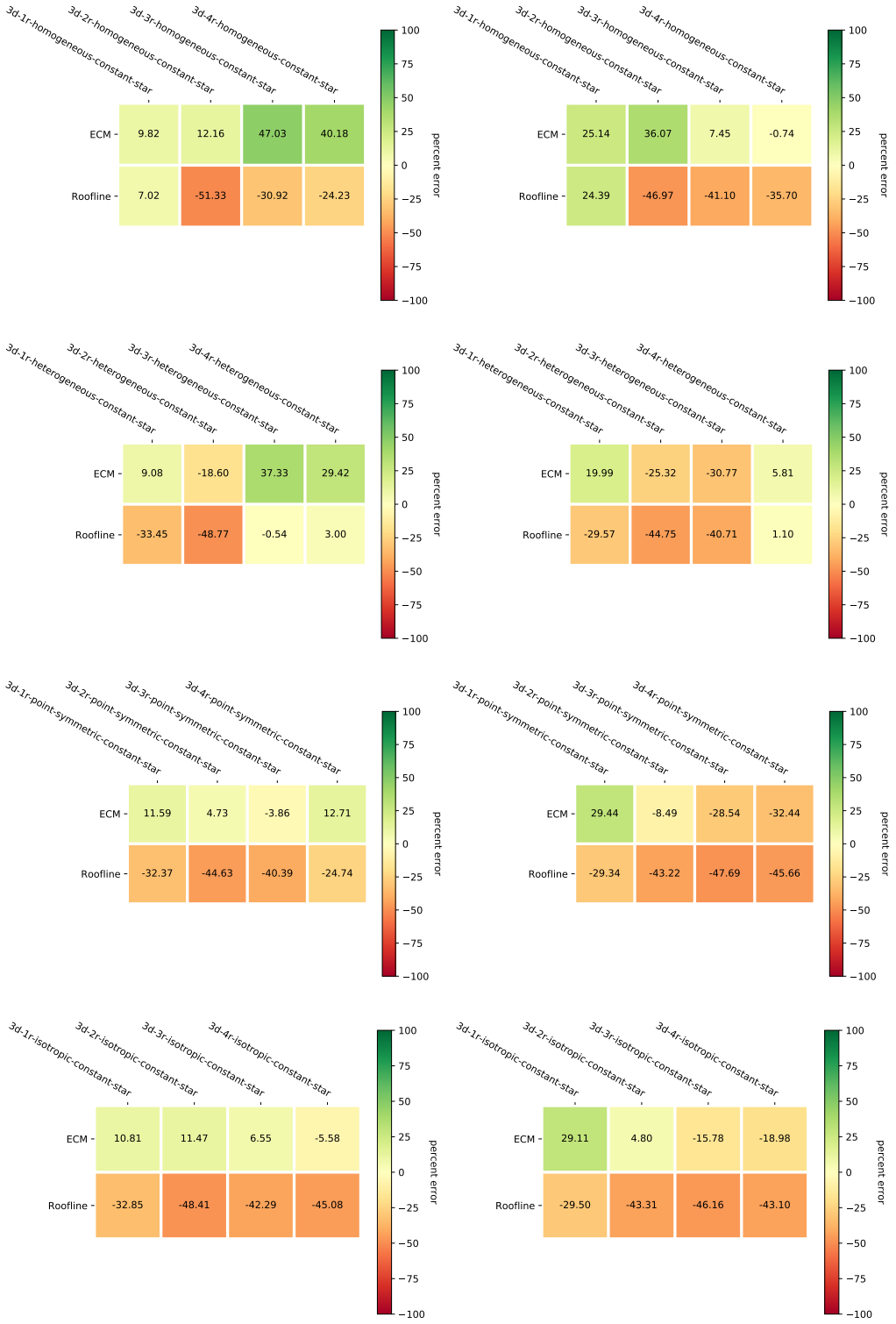

Figure 10.5: *Percentage error of the prediction obtained through ECM and Roofline model, applied to a 3-dimensional star stencil with homogeneous, heterogeneous, point-symmetric and isotropic constant coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through* PROVA!.

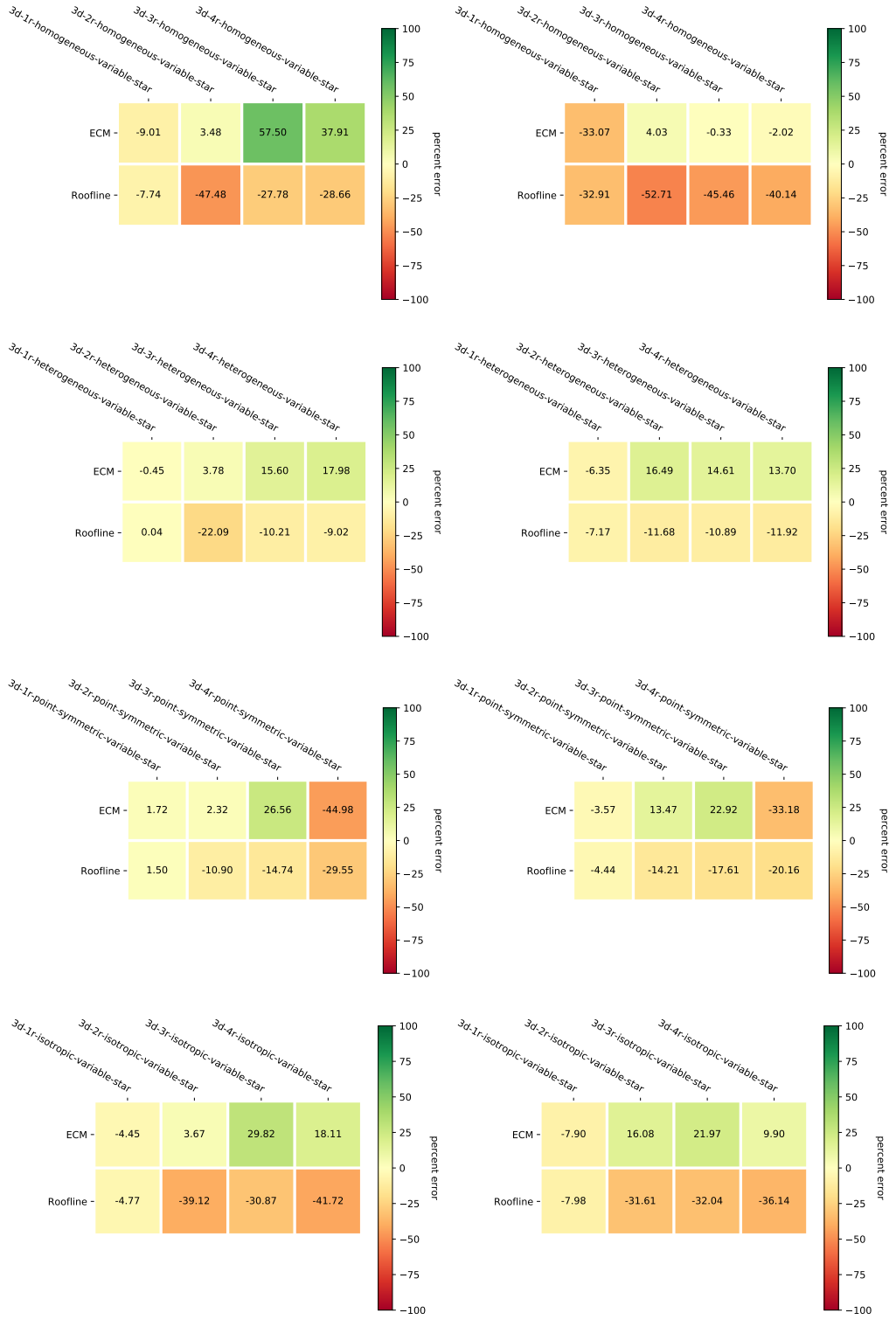

Figure 10.6: *Percentage error of the prediction obtained through ECM and Roofline model, applied to a 3-dimensional star stencil with homogeneous, heterogeneous, point-symmetric and isotropic variable coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through* PROVA!.

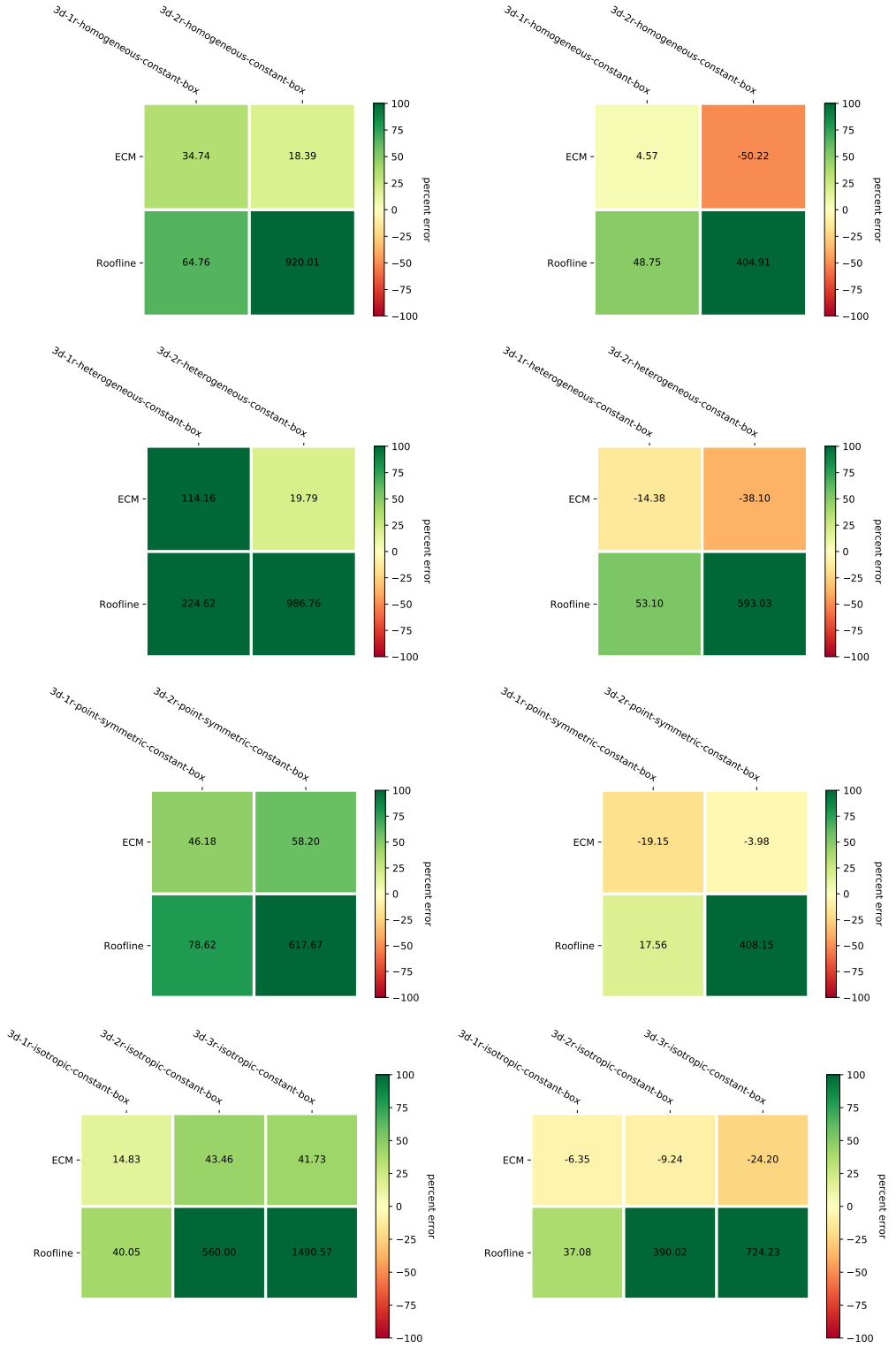

Figure 10.7: *Percentage error of the prediction obtained through ECM and Roofline model, applied to a 3-dimensional box stencil with homogeneous, heterogeneous, point-symmetric and isotropic constant coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through* PROVA!.

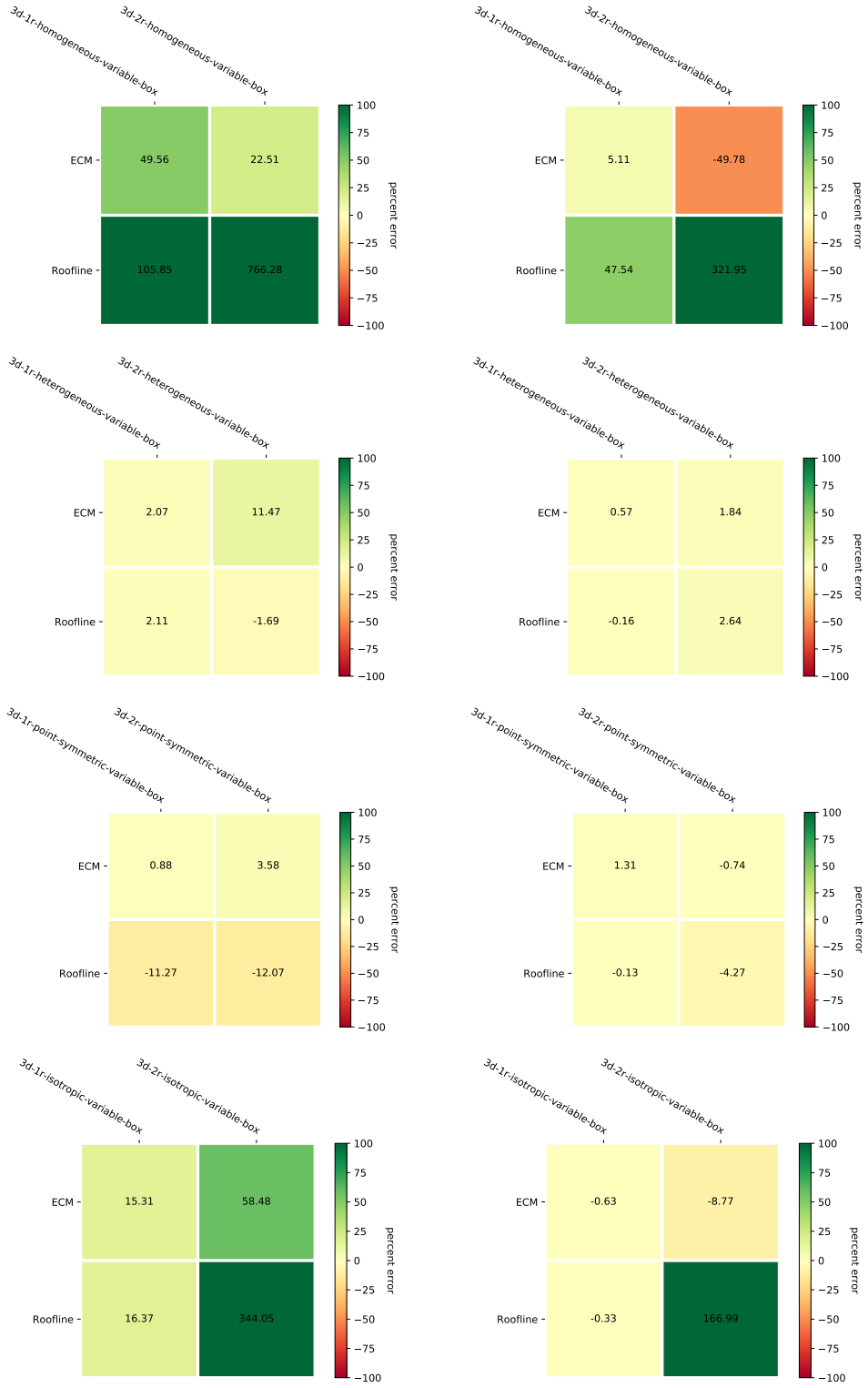

Figure 10.8: *Percentage error of the prediction obtained through ECM and Roofline model, applied to a 3-dimensional box stencil with homogeneous, heterogeneous, point-symmetric and isotropic variable coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through* PROVA!.

it represents a property of the compiler. Only by evaluating a compiler on several architectures it is possible to understand how to improve its usage.

10.2 Discussion of the Results

Having all the experiments being executed using PROVA!, we ensure their reproducibility, as discussed in Chapter 5 and in [87, 59, 56]. We can thus confidently focus on the analysis of the results concerning the performance models used (focus of the validation macro-experiment presented in Section 10.2.1) and to the stencil compilers that generated the source code (the focus of the performance engineering cycle macro-experiment presented in Section 10.2.2).

10.2.1 Validation Experiment

The first macro-experiment has been conceived and performed with the idea of having a big picture of how the Roofline and the ECM performance model predict the execution of the stencil kernels described in Section 9.2.1 on the systems presented in Section 9.1. The methods used have been introduced in Section 9.3.

In Figures 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8 are presented the percentage errors of the predicted performance for a given stencil kernel, obtained using ECM and Roofline models, against the measured performance, obtained executing the experiments through PROVA!. All the predictions (of the performance of a socket) have been obtained through Kerncraft by running a command such as the one presented in Listing 10.1.

```
$ kerncraft -P LC -p Roofline -p ECM stencil.c \
 -m IvyBridgeEP_E5-2660v2.yml -D M 3000 -D N 3000\
 --unit=FLOP/s --cores=10 -vv
```

Listing 10.1: *A possible command used to obtain the performance prediction of Roofline and ECM models for a stencil on an Ivy Bridge E5-2660v2 machine.*

Each measured performance is the average of 10 executions, each running over 10 cores (threads). The quantities measured are: GFLOP/s, GLUP/s, execution time, iterations, FLOP. The percentage error has been

calculated according to the following formula:

$$\%Error = \frac{Predicted - Measured}{Measured} * 100.$$

A negative value of the percentage error thus means that the measured value is higher than the predicted one, and vice-versa.

In this work, when the percent error is smaller than 10%, the predictions are considered fully validated by the experiments.

The color bars vary from red to green, with these extremes meaning a lousy matching between predictions and measured values. Values in the -10% to 10% are acceptable and can be visually recognized by a light green or light orange.

The first comments address the difference between the error of the prediction for the same stencil on the different machines (see Section 4.1). Under this point of view, the results are positive, since the light and dark spots mostly match between the two systems, with some exceptions such as the 2-dimensional isotropic variable star stencil, shown in Figure 10.2.

Generally, measured and predicted results are closer on Emmy (the graphs on the left in Figures 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8). It must be noted that the higher rate of percent error on miniHPC could be caused by an approximation in the modeling of the architecture. Kerncraft accepts as an input, together with a specification of the kernel, also a description of the architecture. MiniHPC differs from Emmy in the ways of associativity of the L3, in particular, said number is not a power of two, which makes it impossible for Kerncraft to handle it. Consequently, the ways of associativity, when describing miniHPC, have been modeled as the closest power of two and the number of sets has been modified accordingly:

$$sets * ways\ of\ associativity * cache\ line\ size = cache\ size.$$

For some kernels, such as 2-dimensional heterogeneous/isotropic constant box stencil, both models fail on both machines. It is an interesting aspect to investigate in more details, and it is the object of the problem described in Section 9.2.2.

By definition, the roofline model represents an upper bound to the achievable performance. Thus its percent error on the heatmaps should always be green (positive). There are some cases when this does not happen, such as a 2-dimensional isotropic variable box (Figure 10.4), and the

3-dimensional stencils such as the isotropic variable star. In the latter ECM matches the measured performance much better than Roofline.

In the calculation of the Roofline, Kerncraft tries to take into account the contention of the shared resources (L3 cache), by equally dividing them among the cores available on the socket. While on one side it is correct to assume such a contention, the original assumption may result too pessimistic: if the bottleneck is the L3, then it may underestimate the predicted performance. In fact, the closest selected benchmark differs between the predicted execution with one core and with ten cores. We believe this to be an aspect to be addressed into the models calculated by Kerncraft.

It is worth noting that the performance measured is obtained through a naive OpenMP parallelization and the threads are pinned to the logical nodes. The strategy used to decide the thread/core affinity also has an impact on the performance, when using the full node, as shown in the next Section.

10.2.2 Performance Engineering Cycle Experiment

Besides the performance analysis, that has been conducted using Kerncraft, and the reproducible performance experiments, thanks to PROVA!, it is highly relevant to be able to interpret the results, both when the measured performance matches the predicted one, and also when it does not. Such an interpretation allows discerning when the model needs to be adapted, from when the code must be optimized, to conclude the performance analysis cycle with the best possible parallel implementation of a given kernel.

The role of this Section is precisely the one to provide a correct interpretation of performance analysis and reproducible experimentation, framed by the taxonomy of experiments described in Section 4.1.

The experiments commented here, i.e., 2-dimensional isotropic box stencil with constant coefficients and radius 1 and 4, have been described in Section 9.2.2, run on a single node of both the clusters described in Section 9.1, without explicitly pinning the threads first and then pinning them to the cores of the node, using the tool Likwid and the pinning strategies illustrated in Section 6.3.

In all our performance graphs, the histograms show the average value, while the error bars show the standard deviation obtained out of 10 executions. The stencil compilers considered (PLUTO and PATUS) use OpenMP

directives for running on a system with multiple threads. Because of this, we compare said approaches against a naive OpenMP implementation (with NUMA aware initialization) of the chosen stencil to evaluate them. We experienced a strong fluctuation in the outputs, concerning performance, when not using explicit pinning.

The heatmap presented in the previous section (Figure 10.3), shows that both ECM and Roofline models almost perfectly match the measured performance on Emmy for the radius one kernel, and they both fail in predicting the performance for the radius four kernel. In the latter case, ECM underestimates the performance on both systems, the prediction being slightly worse on miniHPC; Roofline instead overestimates the performance of a socket, by a factor of 2.5. As explained earlier in Section 10.2.1, in the case of miniHPC there is a representation issue (hardware description) that may cause a higher error than on Emmy, that is well modeled.

Recalling the concepts of access pattern and locality explained in Section 8.2.3 and shown in Figure 8.5, we can derive that when executing the 2-dimensional isotropic box constant stencil with radius 1, and dealing with a *cache size* $> N + 2$ (with N being the number of columns of the grid), only 1 load and 1 store, plus the write allocate take place. In case of the 2-dimensional isotropic box constant stencil with radius 4, when dealing with a *cache size* $> N + 8$, also only requires 1 load and 1 store, plus the write allocate. Thus, under the stated conditions, both stencils only need to transfer 3 data from/to the cache:

$$Data\ traffic = 3 * 8\ bytes = 24B.$$

The number of FLOP to execute is 11 in the case of the radius 1 stencil, thus the arithmetic intensity to the memory is:

$$AI = \frac{1}{B_c} = \frac{11FLOP}{24B} = 0.46.$$

The number of FLOP to execute is 89 in the case of the radius a stencil, thus the arithmetic intensity to the memory is :

$$AI = \frac{1}{B_c} = \frac{89FLOP}{24B} = 3.7.$$

```
=== Roofline ===
Bottlenecks:
  level | a. intensity |   performance   |   peak bw     | peak bw kernel
```

```
--------+--------------+-----------------+---------------+----------------
    CPU |              |  176.00 GFLOP/s |               |
     L1 |  0.59 FLOP/B |  198.23 GFLOP/s |   338.55 GB/s | triad
     L2 |   1.0 FLOP/B |  328.58 GFLOP/s |   324.89 GB/s | triad
     L3 |   3.7 FLOP/B |  892.56 GFLOP/s |   240.69 GB/s | copy
    MEM |   3.7 FLOP/B |  148.07 GFLOP/s |    39.93 GB/s | copy

Cache or mem bound.
148.07 GFLOP/s due to MEM transfer bottleneck (with bw from copy benchmark)
Arithmetic Intensity: 3.71 FLOP/B

=== ECM ===
T_nOL = 360.0 cy/CL
T_OL = 640.0 cy/CL
L2 = 71.20 GFLOP/s
L3 = 261.07 GFLOP/s
MEM = 150.74 GFLOP/s
memory cycles based on copy kernel with 40.65 GB/s
{ 640.0 || 360.0 | 22.0 | 6.0 | 10.4 } cy/CL = 2.45 GFLOP/s
{ 640.0 \ 640.0 \ 640.0 \ 640.0 } cy/CL
saturating at 61.6 cores
```

Listing 10.2: *ECM prediction through Kerncrat of a 2-dimensional, radius 4, isotropic box stencil with constant coefficients and size* 3000^2 *on Emmy.*

```
=== Roofline ===
Bottlenecks:
  level | a. intensity |   performance   |   peak bw     | peak bw kernel
--------+--------------+-----------------+---------------+----------------
    CPU |              |  416.00 GFLOP/s |               |
     L1 |  0.59 FLOP/B |  227.21 GFLOP/s |   388.04 GB/s | triad
     L2 |   1.0 FLOP/B |  384.95 GFLOP/s |   380.62 GB/s | triad
     L3 |   3.7 FLOP/B | 1232.20 GFLOP/s |   332.28 GB/s | copy
    MEM |   3.7 FLOP/B |  236.63 GFLOP/s |    63.81 GB/s | copy

Cache or mem bound.
227.21 GFLOP/s due to L1 transfer bottleneck (with bw from triad benchmark)
Arithmetic Intensity: 0.59 FLOP/B

=== ECM ===
T_nOL = 360.0 cy/CL
T_OL = 640.0 cy/CL
L2 = 84.15 GFLOP/s
L3 = 308.53 GFLOP/s
MEM = 236.96 GFLOP/s
memory cycles based on copy kernel with 63.90 GB/s
{ 640.0 || 360.0 | 22.0 | 6.0 | 7.8 } cy/CL = 2.89 GFLOP/s
{ 640.0 \ 640.0 \ 640.0 \ 640.0 } cy/CL
saturating at 81.9 cores
```

Listing 10.3: *ECM prediction through Kerncrat of a 2-dimensional, radius 4, isotropic box stencil with constant coefficients and size* 3000^2 *on miniHPC.*

```
=== Roofline ===
Bottlenecks:
```

```
  level | a. intensity |   performance   |   peak bw     | peak bw kernel
--------+--------------+-----------------+---------------+----------------
    CPU |              |  176.00 GFLOP/s |               |
     L1 |  0.59 FLOP/B |  198.23 GFLOP/s |   338.55 GB/s | triad
     L2 |   1.0 FLOP/B |  328.58 GFLOP/s |   324.89 GB/s | triad
     L3 |   3.7 FLOP/B |  892.56 GFLOP/s |   240.69 GB/s | copy
    MEM |   3.7 FLOP/B |  148.07 GFLOP/s |    39.93 GB/s | copy

Cache or mem bound.
148.07 GFLOP/s due to MEM transfer bottleneck (with bw from copy benchmark)
Arithmetic Intensity: 3.71 FLOP/B

=== ECM ===
T_nOL = 360.0 cy/CL
T_OL = 640.0 cy/CL
L2 = 71.20 GFLOP/s
L3 = 261.07 GFLOP/s
MEM = inf YFLOP/s
memory cycles based on load kernel with 46.20 GB/s
{ 640.0 || 360.0 | 22.0 | 6.0 | 0.0 } cy/CL = 2.45 GFLOP/s
{ 640.0 \ 640.0 \ 640.0 \ 640.0 } cy/CL
saturating at inf cores
```

Listing 10.4: *ECM prediction through Kerncrat of a 2-dimensional, radius 4, isotropic box stencil with constant coefficients and size* 500^2 *on Emmy.*

```
=== Roofline ===
Bottlenecks:
  level | a. intensity |   performance   |   peak bw     | peak bw kernel
--------+--------------+-----------------+---------------+----------------
    CPU |              |  416.00 GFLOP/s |               |
     L1 |  0.59 FLOP/B |  227.21 GFLOP/s |   388.04 GB/s | triad
     L2 |   1.0 FLOP/B |  384.95 GFLOP/s |   380.62 GB/s | triad
     L3 |   3.7 FLOP/B | 1232.20 GFLOP/s |   332.28 GB/s | copy
    MEM |   3.7 FLOP/B |  236.63 GFLOP/s |    63.81 GB/s | copy

Cache or mem bound.
227.21 GFLOP/s due to L1 transfer bottleneck (with bw from triad benchmark)
Arithmetic Intensity: 0.59 FLOP/B


=== ECM ====
T_nOL = 360.0 cy/CL
T_OL = 640.0 cy/CL
L2 = 84.15 GFLOP/s
L3 = 308.53 GFLOP/s
MEM = inf YFLOP/s
memory cycles based on load kernel with 57.39 GB/s
{ 640.0 || 360.0 | 22.0 | 6.0 | 0.0 } cy/CL = 2.89 GFLOP/s
{ 640.0 \ 640.0 \ 640.0 \ 640.0 } cy/CL
saturating at inf cores
```

Listing 10.5: *ECM prediction through Kerncrat of a 2-dimensional, radius 4, isotropic box stencil with constant coefficients and size* 500^2 *on miniHPC*

Such calculations are confirmed by the grey box analysis performed using Kerncraft (running a command such as the one presented in List-

ing 10.1) and displayed in Listings 10.2 and 10.3.

The automatic analysis of Kerncraft identifies a different bottleneck on Emmy (memory) and miniHPC (L1), thus resulting in the bottleneck, not in the L3 to memory data path, but in L1. The bottleneck contributes to the Roofline prediction that, in both cases, overestimates the expected performance: a prediction of 148.7 GFLOP/s on Emmy (against the measured value of 44.52 GFLOP/s) and 227.21 GFLOP/s (against the measured value of 66.7 GFLOP/s) on miniHPC.

Instead, the ECM model predicts a saturation point greater than the socket, but an expected performance per core quite low, that brings to a forecast of 24.5 GFLOP/s on Emmy (against the measured value of 44.52 GFLOP/s) and 28.9 GFLOP/s (against the measured value of 66.7 GFLOP/s) on miniHPC.

Commenting on the overestimation of the performance of the Roofline model, it is worth noting that it may mean that there is an available performance to be exploited, that the actual code does not, which should be addressed in the optimization phase.

Similar results are measured and predicted for the 2-dimensional box stencil with isotropic constant coefficients, radius 4 and grid size 500^2, with the difference that, in this case, the grid is small enough to fit in cache. The Roofline prediction overestimates the expected performance: a prediction of 148.7 GFLOP/s on Emmy (against the measured value of 43.99 GFLOP/s) and 227.21 GFLOP/s (against the measured value of 63.08 GFLOP/s) on miniHPC. The ECM model instead predicts a saturation point greater than the socket, but an expected performance per core quite low, that brings to a forecast of 24.5 GFLOP/s on Emmy (against the measured value of 43.99 GFLOP/s) and 28.9 GFLOP/s (against the measured value of 66.7 GFLOP/s) on miniHPC.

Switching to the 2-dimensional isotropic box constant stencil with radius 1, the measured values for a grid size of 500^2 are 29.39 GFLOP/s and 73.18 GFLOP/s on Emmy and miniHPC respectively. The ECM prediction has been 64.5 GFLOP/s on Emmy and 67.7 GFLOP/s on miniHPC, while the Roofline prediction has been 18.30 on Emmy and 29.25 GFLOP/s on miniHPC. Said predictions differ quite a lot and the Roofline ones have been proven to be extremely wrong. Identical comments can be done for the version using a 3000^2 grid.

Some considerations related to the compilers are also needed. Stencils compilers have the task to parallelize a stencil computation automatically, often accepting a pseudo-code (a DSL in the case of PATUS) or plain,

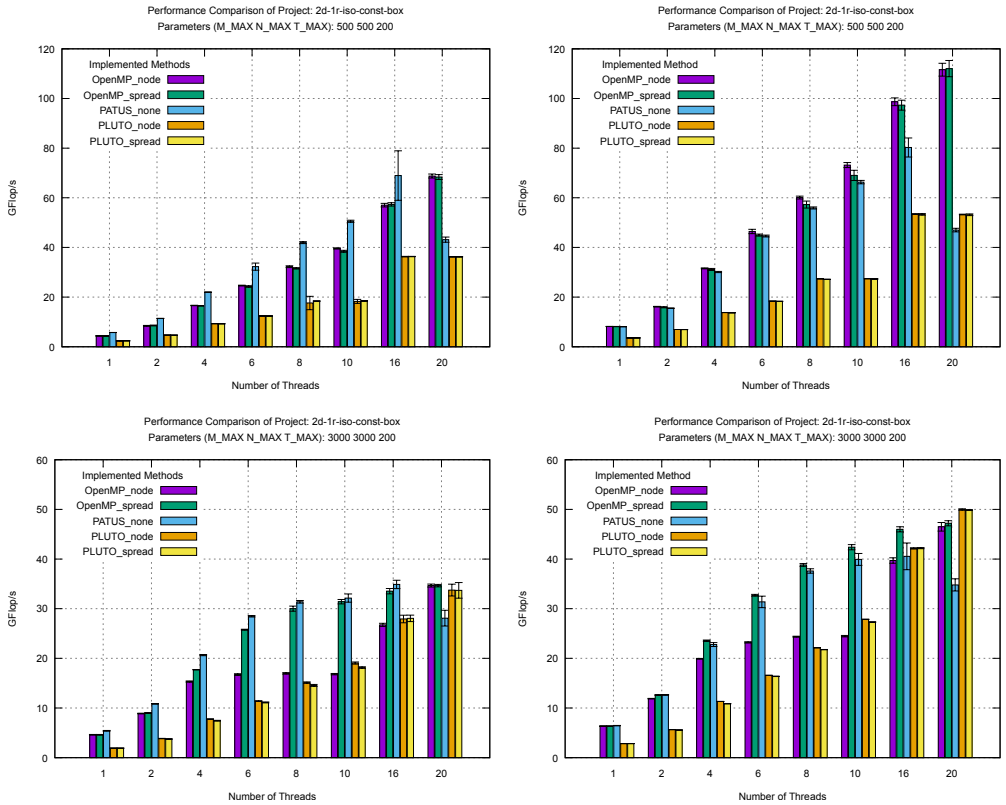

Figure 10.9: *Performance graph of a 2D isotropic, box stencil with constant coefficients and radius 1. Two dimensions for the grids have been used:* 500^2 *and* 3000^2*, while the timesteps are 200. On the left are presented the results obtained on Emmy, on the right the ones on miniHPC. The kernel has been implemented using three methods: naive OpenMP implementation with explicit pinning using the strategies ByNode and BySpreading,* PATUS *without pinning, and* PLUTO *with explicit pinning using the strategies ByNode and BySpreading. For all the graphs the histogram shows the average value out of 10 executions, and the error bars the standard deviation.*

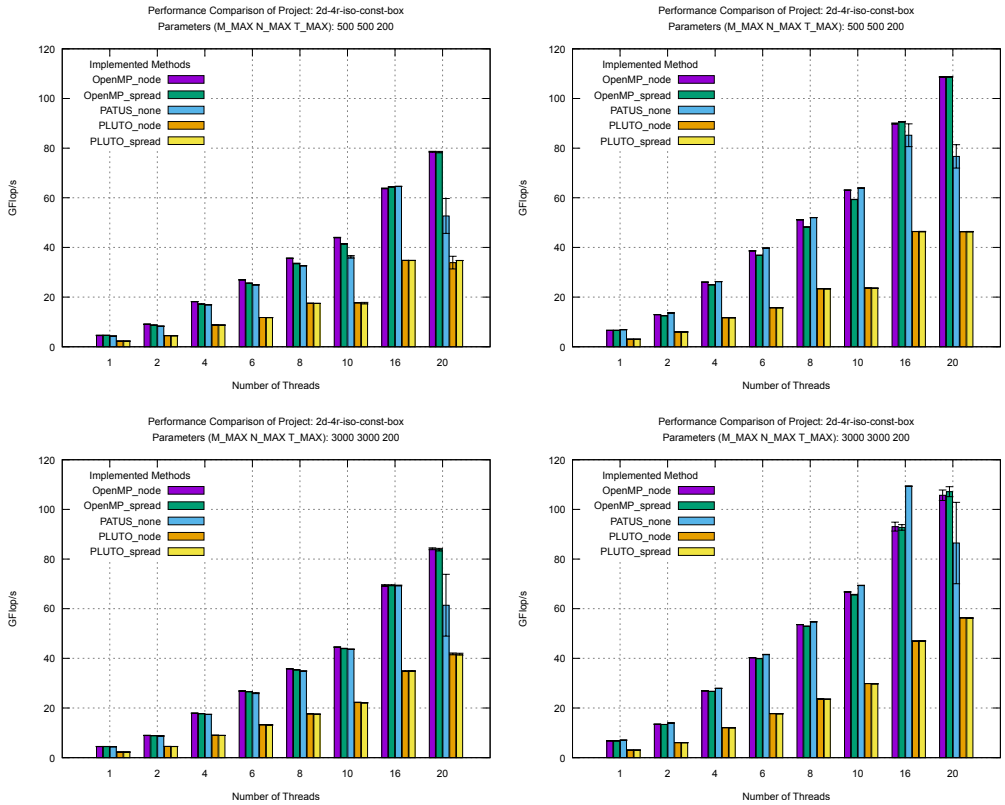

Figure 10.10: *Performance graph of a 2D isotropic, box stencil with constant coefficients and radius 4. Two dimensions for the grids have been used:* 500^2 *and* 3000^2*, while the timesteps are 200. On the left are presented the results obtained on Emmy, on the right the ones on miniHPC. The kernel has been implemented using three methods: naive OpenMP implementation with explicit pinning using the strategies ByNode and BySpreading,* PATUS *without pinning, and* PLUTO *with explicit pinning using the strategies ByNode and BySpreading. For all the graphs the histogram shows the average value out of 10 executions, and the error bars the standard deviation.*

unoptimized C code (e.g., in the case of PLUTO): they apply optimization strategies to generate an output source code that then needs to be compiled. PLUTO makes use of the diamond tiling technique to optimize the performance of the stencil automatically. Of course, tiling offers benefits when the number of operations to execute is limited, thus resulting, ideally, in better performance gains when applied to stencils with a low number of neighbors. This is precisely what has been experienced in our experiments, where PLUTO offers excellent performance for the radius one stencil considered, for both grid sizes (see Figure 10.9). It instead struggles to produce good code in the case of the radius four stencil (see Figure 10.10). Having conducted experiments varying the grid size used as an input turned out to be an excellent decision to have a better overall understanding, which could lead to the use of a different compiler when changing the stencil size.

PATUS that "aims to provide a means towards productivity and performance" [27], performs very decently, producing almost in each case a good code (concerning the obtained performance) that scales linearly when increasing the number of threads for both stencils used and both the sizes of the input grid. The experiments highlight an issue whit its code generated to use a full node, i.e., with 20 threads: we always experienced a considerable drop in performance, with a high value of the standard deviation (see Figures 10.9 and 10.10). It is worth noting that PATUS exploits the auto-tuning technique to optimize parameters like *chunk size* and *unrolling level*: for the cases analyzed in this experiment, it failed in its task.

Both in Figures 10.9 and 10.10 the behavior of OpenMP is interesting: the histogram in purple represents the performance with pinning to the logical core (thus using first fully one socket and then passing to the next one) while the histogram in green represents its performance when pinning the threads by spreading them between the available sockets. As known in theory, and described in Section A.2 of [62], memory bound codes benefit when the threads are spread across all sockets. Such an effect can clearly be seen for the radius one stencil, in Figure 10.9, where the performance hits, both on Emmy and miniHPC, the limit of the socket: a saturation at 4.3 cores has been predicted by the ECM model, for the stencil using a 3000^2 grid. Saturation happens around six cores, whereas if using a spreading strategy for pinning, the performance keeps scaling. When executing with enough threads to utilize both the sockets, even when pinning logically to the cores, then the performance of the

two strategies tend to be equivalent, reaching the limit of the architecture. Considering the same stencil (radius 1), but using a grid of dimension 500^2 it is not possible to appreciate this effect since the grid is small enough to fit in cache, thus not needing frequent data movement from memory.

Part IV

Conclusions & Future Work

Chapter 11

Conclusions and Future Work

In this thesis has been presented the problem of reproducibility in the context of High Performance Computing, by describing the issues that arise with the increased complexity of modern systems both at the hardware and the software level. It is crucial to parallelize the programs, often via hardware specific optimizations and configurations, and sophisticated program transformations that may hinder the reproducibility of their results and performance.

Following a description of the factors affecting the reproducibility, a taxonomy of the experiments in Computational Sciences has been introduced, serving to qualify the information necessary for successful reproduction. Varying the experiments, it is possible to target (and achieve) any level of reproducibility: repetition, replication, and re-experimentation. The description of an experiment, including the configurations and customizations needed to execute it correctly, is part of a framework to conduct reproducible research.

A possible implementation of such a framework, PROVA! [57] has been shown and then used to conduct the experiments that have been discussed in this work, thus ensuring their reproducibility. PROVA! interfaces with tools that help to address the reproducible building and installation of software, the configuration of the environment, thus providing a reproducible software stack and environment at the time of executing an experiment, contributing to its overall reproducibility. In addition to that, PROVA! maintains code and description of the experiments, providing documentation, thus enabling their dissemination, acting as an artifact description.

The experiments presented in this work are related to thr performance engineering of stencil kernels. Stencil computation belongs to the structured grid motif of the Dwarfs of Berkeley [11]. It is a relevant pattern in scientific computing since it appears in applications ranging from the weather forecast to geophysics, computational fluid dynamics, and image processing. The stencil pattern has been characterized and described using keywords related to shape, dimensionality, radius, and properties of its coefficients.

A tool to generate stencils kernels has been implemented and interfaced to Kerncraft, thus enabling the modeling of their performance. In this way it is possible to perform experimentation in the complex topic of Performance Engineering: evaluating the expected performance of a code is needed to verify the bottlenecks that it may hit when running on a specific architecture. Performance modeling can be carried out either by analytic modeling of the machine and the code, or by using grey-box tools like Kerncrat [64], capable of applying the Roofline [135] and ECM [61] models. The results of the modeling, i.e., predicted performance, is then validated against the reproducible performance results obtained by executing the code through PROVA!: in case of a mismatch, either the model applied must be adapted, or the code optimized and tuned to exploit the hardware it runs on. An overview of the behavior of the performance models is given when running experiments involving a wide variety of stencil kernels on two different machines.

An in-depth analysis is shown for a selected kernel, also by varying the input size to fit or not into the cache. In such a case the kernel has been implemented by using three different approaches: a state of the art solution (OpenMP) that offers control and very good performance, but needs expertise; a solution that allows for automatic parallelization by inserting a simple directive that states the region of the code to be optimized by the PLUTO compiler; a solution, via PATUS, that offers to fill the gap between performance and programmability, by defining a stencil in a high-level domain-specific language that is internally translated into optimized code, transparently to the user. Such a test case provided the possibility to evaluate not only the performance models but also such alternative implementations in different working conditions, shedding light on the performance-programmability trade-off of the modern architectures.

The application of the proposed framework for reproducibility, and the use of PROVA!, together with grey box modeling tools (as demon-

strated in the experiments), ideally traces a path towards a discipline of performance engineering.

11.1 Contributions and Relevance to the Community

The contributions of this work are manifold:

- Definition of a Taxonomy of Experiments in Computational Sciences
- Definition of three Levels of Reproducibility, based on the experimental setup, that allow classifying the status of a research, channeling it towards credibility and thus re-use of work, which is one of the cornerstones of science
- A framework to conduct reproducible research in Computational Sciences and its implementation as a distributed workflow and system management tool, PROVA!
- Stencils generator
- Stencil performance analysis through grey-box modeling and validation
- Application of the proposed framework to the sub-field of structured grids, e.g. stencil motif, on different architectures, obtaining a fair comparative analysis of the performance of stencil kernels, produced using stencil compilers
- As a whole, the previous contributions open the path to a Discipline of Performance Engineering

The relevance to the community is proven by the following publications, strictly related to this work:

- Reproducible Experiments in Parallel Computing: Concepts and Stencil Compiler Benchmark Study. Danilo Guerrera, Helmar Burkhart, and Antonio Maffia.In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 464–474. Springer International Publishing, 2014 [55].

- Trusted High-Performance Computing in the Classroom. Helmar Burkhart, Danilo Guerrera, and Antonio Maffia. 2014 *Workshop on Education for High Performance Computing*, pages 27–33, 2014 [20].

- Reproducibility in Practice: Lessons Learned from Research and Teaching Experiments. Antonio Maffia, Helmar Burkhart, and Danilo Guerrera. In *Euro-Par 2015: Parallel Processing Workshops*. Springer International Publishing, 2015 [87].

- No more Believe Me: Make Your Informatics Experiments Reproducible. Helmar Burkhart, Danilo Guerrera, and Antonio Maffia [21].

- Reproducible Stencil Compiler Benchmarks Using PROVA!. Danilo Guerrera, Helmar Burkhart, and Antonio Maffia. In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 108–115. IEEE, 2016 [56].

- Reproducible stencil compiler benchmarks using PROVA!. Danilo Guerrera, Antonio Maffia, and Helmar Burkhart. *Future Generation Computer Systems*, vol. 92, pages 933-946, may 2018 [59].

11.2 Future Work

The work presented in this dissertation represents only the first steps, raising several problematics that need to be addressed in the future.

The taxonomy of the experiments that has been introduced has been thought of having in mind the perspective of high performance computing experiments, thus focusing on the performance. It may need to be verified and possibly extended if it proves to be not adequate enough in a specific niche-domain.

The topic of reproducibility is dominant at the time of writing, but there is no comprehensive and universally agreed terminology: such an issue may lead to misunderstandings and could result in a significant obstacle to the diffusion to the broad public of scientific researchers in the HPC field. Works directly addressing this issue have been recently published ([15], [103], [114], [120], [13]), and the focus to it given by major conferences, such as the "International Conference for High Performance Computing, Networking, Storage and Analysis" (Supercomputing), may

help to accelerate the convergence towards a widely agreed nomenclature.

PROVA!, a possible implementation of the proposed framework for reproducibility, is explicitly targeting experiments related to stencil computation. In [59] it has been proved to be flexible enough to conduct experiments focusing on a different kind of problems and adopting a different paradigm, such as GPU computation and MPI related experiments.

Some limits emerged in our approach since there are factors that cannot be easily controlled. Obviously, the hardware itself is an uncontrollable entity and the OS, for instance, provides some essential libraries that are used as a basis. Thus, a different OS represents a distinct starting point on top of which the software stack is built. Additionally, some applications, such as machine learning based codes or, as in one of our test cases, auto-tuning of parameters, may have a non-deterministic nature. Furthermore, even when using EasyBuild's installation recipes on several machines, it may seldom happen that issues appear thus leading to an irreproducible build. While this is usually not the case, it has occurred, mostly due to OS libraries or BIOS level configurations. While it is not possible to avoid the hardware issue, we can extend our control of the software by using containerization solutions, such as Shifter [49] and Singularity [81]. Those would represent an alternative solution to the combination of Lmod and EasyBuild, that needs to be explored.

The stencils categorization can be further refined to include alternative patterns, considering: different input and output grids, dependence on more than just the previous time-step (time-dependence), alternative sweeps, such as Gauss-Seidel iterations or red-black Gauss-Seidel. Furthermore, the performance models and the codes should be validated in the presence of non-quadratic and non-cubic grids.

Already having experience with using and porting PROVA!, future work could include providing collaboration support via GitHub. To do so, collaboration and provenance issues must be addressed. On a technical side, a goal is to build a distributed workspace system using technologies like git, to provide a shared workspace across several parallel machines and supercomputing sites, as well as a way to collaborate to a joint project.

The predictions obtained applying Roofline, and ECM models should always be used to validate the measured performance and vice-versa, resulting in a "Gold Standard" for performance engineering. In this work, this has been achieved by using STEMPEL and its interface to PROVA!:

such a workflow should be integrated into PROVA! to make it easily accessible, helping to build a community of reproducible performance engineering and strengthen its collaborative ecosystem.

The experiments presented in this work, have been conducted in two small laboratories: it is needed to confirm and expand both solutions and analysis to supercomputers, which possibly consist of hybrid solutions at the node level.

If we were to trust the computational results, thus allowing re-use of work and the option to build on top of existing solutions obtained by other scientists, reproducibility needs to be emphasized and more widely applied. On the technical side, there is a need for tools and platforms that allow executing, storing and remotely accessing reproducible experiments, and offer collaboration support for team efforts. As we proposed in [20], on the social side, it may be thought of incentives for those who voluntarily spend time in making their experiments reproducible and must be developed teaching strategies for making the next generation of computer scientists aware of the relevance and importance of reproducibility aspects.

Bibliography

[1] Cpu db, a complete database of processors for researchers and hobbyists alike. http://cpudb.stanford.edu. [cited at p. 16, 175]

[2] Linux cpufreq. https://kernel.org/doc/Documentation/cpu-freq/governors.txt. [cited at p. 27]

[3] Skampy: Ultra-skalierbare multiphysik-simulationen für erstarrungsprozesse in metallen. http://www.walberla.net/fundingskampy.html. [cited at p. 77]

[4] Transistor count. https://www.revolvy.com/page/Transistor-count. [cited at p. 17, 175]

[5] HPL - A portable implementation of the high-performance Linpack Benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/, 2008. [cited at p. 50]

[6] Onlinehpc. http://www.onlinehpc.com, 2012. Accessed: 2015-04-30. [cited at p. 11]

[7] Intel® architecture code analyzer. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer, 2017. Accessed: 2018-08-09. [cited at p. 92]

[8] Open source architecture code analyzer. https://github.com/RRZE-HPC/OSACA, 2018. Accessed: 2018-08-09. [cited at p. 92]

[9] Top500, the list. https://www.top500.org, jun 2018. [cited at p. 34, 44, 85]

[10] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. [cited at p. 36, 176]

[11] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. [cited at p. 75, 87, 136]

[12] David H. Bailey, Jonathan M. Borwein, and Victoria Stodden. Facilitating reproducibility in scientific computing: Principles and practice. In *Reproducibility: Principles, Problems, Practices, and Prospects*, pages 205–231. John Wiley & Sons, Inc., jul 2016. [cited at p. 8]

[13] Lorena A. Barba. Terminologies for reproducible research. *CoRR*, abs/1802.03311, 2018. [cited at p. 7, 48, 49, 138]

[14] C. Glenn Begley and Lee M. Ellis. Drug development: Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, Mar 2012. [cited at p. 7, 8, 49]

[15] Fabien C. Y. Benureau and Nicolas P. Rougier. Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions. *Frontiers in Neuroinformatics*, 11, jan 2018. [cited at p. 138]

[16] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. Knime: The konstanz information miner. In Christine Preisach, Hans Burkhardt, Lars Schmidt-Thieme, and Reinhold Decker, editors, *Data Analysis, Machine Learning and Applications*, pages 319–326, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. [cited at p. 10, 48, 49]

[17] U. Bondhugula, V. Bandishti, and I. Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, May 2017. [cited at p. 82]

[18] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM. [cited at p. 82, 96]

[19] Eric L. Boyd, Waqar Azeem, Hsien-Hsin Lee, Tien-Pao Shih, Shih-Hao Hung, and Edward S. Davidson. A hierarchical approach to modeling and improving the performance of scientific applications on the ksr1. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume*

03, ICPP '94, pages 188–192, Washington, DC, USA, 1994. IEEE Computer Society. [cited at p. 87]

[20] Helmar Burkhart, Danilo Guerrera, and Antonio Maffia. Trusted high-performance computing in the classroom. *2014 Workshop on Education for High Performance Computing*, pages 27–33, 2014. [cited at p. 138, 140]

[21] Helmar Burkhart, Danilo Guerrera, and Antonio Maffia. No more believe me : Make your informatics experiments reproducible. http://www.informatics-europe.org/images/ECSS/ECSS2015/ECCS2015-Burkhart.pdf, 2015. [cited at p. 138]

[22] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst.*, 16(6):1768–1810, 1994. [cited at p. 93]

[23] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos. *J. Parallel Distrib. Comput.*, 74(12):3202–3216, December 2014. [cited at p. 38]

[24] A. Casadevall and F. C. Fang. Reproducible science. *Infection and Immunity*, 78(12):4972–4975, sep 2010. [cited at p. 6, 48]

[25] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, aug 2007. [cited at p. 38]

[26] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, May 2011. [cited at p. 83, 96]

[27] Matthias-Michael Christen. *Generating and auto-tuning parallel stencil codes*. PhD thesis, 2011. http://edoc.unibas.ch/diss/DissB9723. [cited at p. 130]

[28] Jon F. Claerbout and Martin Karrenbach. Electronic documents give reproducible research a new meaning. In *SEG Technical Program Expanded Abstracts 1992*. Society of Exploration Geophysicists, jan 1992. [cited at p. 6, 48]

[29] Christian Collberg, Todd Proebsting, Gina Moraila, Zuoming Shi, and Alex M. Warren. Measuring Reproducibility in Computer Systems Research. http://reproducibility.cs.arizona.edu/tr.pdf, 2014. [cited at p. 10, 48]

[30] Christian Collberg, Todd Proebsting, and Alex M Warren. Repeatability and benefaction in computer systems research. Technical Report 14-04, University of Arizona, 2015. [cited at p. 10]

[31] A. P. Davison, M. Mattioni, D. Samarkanov, and B. Teleńczuk. Sumatra: A Toolkit for Reproducible Research. In V. Stodden, F. Leisch, and R. D. Peng, editors, *Implementing Reproducible Research*, chapter 3, pages 57–78. Chapman & Hall, 2014. [cited at p. 11, 48, 49]

[32] A.P. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science Engineering*, 14(4):48–56, July 2012. [cited at p. 50]

[33] Augusto Born de Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. Datamill: Rigorous performance evaluation made easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 137–148, New York, NY, USA, 2013. ACM. [cited at p. 11, 48]

[34] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005. [cited at p. 9]

[35] Oxford English Dictionary. "art, n.1". http://www.oed.com. [cited at p. 6]

[36] M. Dolfi, Jan Gukelberger, Andreas Hehn, J. Imriska, K. Pakrouski, T. F. Rønnow, Matthias Troyer, I. Zintchenko, Fernando Seabra Chirigati, Juliana Freire, and Dennis Shasha. A model project for reproducible papers: critical temperature for the Ising model on a square lattice. *CoRR*, abs/1401.2000, 2014. [cited at p. 49]

[37] Jack Dongarra. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, UK, 1988. Springer-Verlag. [cited at p. 85]

[38] Chris Drummond. Replicability is not reproducibility: Nor is it good science. In *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, 2009. [cited at p. 6, 48]

[39] A. Dubrow. Lmod: The "secret sauce" behind module management at tacc. http://www.tacc.utexas.edu/news/feature-stories/2012/lmod. [cited at p. 43]

[40] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, sep 1972. [cited at p. 25, 175]

[41] Association for Computing Machinery. Artifact review badging. https://www.acm.org/publications/policies/artifact-review-badging, 2018. [cited at p. 11, 48, 51]

[42] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994. [cited at p. 34, 38]

[43] J. Freire and C. T. Silva. Making computations and publications reproducible with vistrails. *Computing in Science Engineering*, 14(4):18–25, July 2012. [cited at p. 9, 48, 49]

[44] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM. [cited at p. 82, 97]

[45] John L. Furlani. Modules : Providing a flexible user environment. In *Proceeding of the Fifth Large Installation System Administration*, pages 141–152, 1991. [cited at p. 42]

[46] John L. Furlani and Peter W. Osel. Abstract yourself with modules. In *Proceedings of the 10th USENIX Conference on System Administration*, LISA '96, pages 193–204, Berkeley, CA, USA, 1996. USENIX Association. [cited at p. 42]

[47] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. The Spack package manager. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*. ACM Press, 2015. [cited at p. 44]

[48] Markus Geimer, Kenneth Hoste, and Robert McLay. Modern scientific software management using easybuild and lmod. In *Proceedings of the First International Workshop on HPC User Support Tools*, HUST '14, pages 41–51, Piscataway, NJ, USA, 2014. IEEE Press. [cited at p. 43, 44, 56, 63, 67, 100]

[49] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. Shifter: Containers for hpc. *Journal of Physics: Conference Series*, 898(8):082021, 2017. [cited at p. 139]

[50] B. Giardine. Galaxy: A platform for interactive large-scale genome analysis. *Genome Research*, 15(10):1451–1455, sep 2005. [cited at p. 9, 48, 49]

[51] Tristan Glatard, Lindsay B. Lewis, Rafael Ferreira da Silva, Reza Adalat, Natacha Beck, Claude Lepage, Pierre Rioux, Marc-Etienne Rousseau, Tarek Sherif, Ewa Deelman, Najmeh Khalili-Mahani, and Alan C. Evans.

Reproducibility of neuroimaging analyses across operating systems. *Frontiers in Neuroinformatics*, 9, apr 2015. [cited at p. 42]

[52] Carole A. Goble, Jiten Bhagat, Sergejs Aleksejevs, Don Cruickshank, Danius Michaelides, David Newman, Mark Borkum, Sean Bechhofer, Marco Roos, Peter Li, and David De Roure. myExperiment: a repository and social network for the sharing of bioinformatics workflows. *Nucleic Acids Research*, 38(suppl_2):W677–W682, may 2010. [cited at p. 9]

[53] Ed H. B. M. Gronenschild, Petra Habets, Heidi I. L. Jacobs, Ron Mengelers, Nico Rozendaal, Jim van Os, and Machteld Marcelis. The effects of FreeSurfer version, workstation type, and macintosh operating system version on anatomical volume and cortical thickness measurements. *PLoS ONE*, 7(6):e38234, jun 2012. [cited at p. 42]

[54] Danilo Guerrera. Source code for the experiments presented in the ph.d. thesis, 2018. https://github.com/sguera/PhDThesisExperiments. [cited at p. 100]

[55] Danilo Guerrera, Helmar Burkhart, and Antonio Maffia. Reproducible experiments in parallel computing: Concepts and stencil compiler benchmark study. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 464–474. Springer International Publishing, 2014. [cited at p. 11, 48, 51, 137]

[56] Danilo Guerrera, Helmar Burkhart, and Antonio Maffia. Reproducible stencil compiler benchmarks using prova! In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 108–115. IEEE, 11 2016. [cited at p. 121, 138]

[57] Danilo Guerrera, Antonio Maffia, and Helmar Burkhart. Prova! try, prove, convince. https://prova.io, 2016. [cited at p. 11, 135]

[58] Danilo Guerrera, Antonio Maffia, and Helmar Burkhart. Source code for fcgs17 experiments, 2017. https://github.com/sguera/FGCS17. [cited at p. 56]

[59] Danilo Guerrera, Antonio Maffia, and Helmar Burkhart. Reproducible stencil compiler benchmarks using PROVA! *Future Generation Computer Systems*, may 2018. DOI: 10.1016/j.future.2018.05.023. [cited at p. 31, 48, 56, 68, 69, 70, 71, 121, 138, 139, 176]

[60] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, may 1988. [cited at p. 37, 176]

[61] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multicore chips via simple machine models. *CoRR*, abs/1208.2908, 2012. [cited at p. 87, 89, 136]

[62] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman and Hall / CRC computational science series. CRC Press, 2011. [cited at p. 22, 28, 93, 94, 130]

[63] Julian Hammer. Layer conditions. https://rrze-hpc.github.io/layer-condition/. [cited at p. 92]

[64] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Kerncraft: A tool for analytic performance modeling of loop kernels. *CoRR*, abs/1702.04653, 2017. [cited at p. 71, 88, 92, 136]

[65] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5 edition, 2012. [cited at p. 27]

[66] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec 1989. [cited at p. 93]

[67] Roger W Hockney and Ian J Curington. f12: A parameter to characterize memory and communication bottlenecks. *Parallel Computing*, 10(3):277 – 286, 1989. [cited at p. 86, 87]

[68] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications*, 14(4):330–346, 2000. [cited at p. 87]

[69] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt. Easybuild: Building software with ease. In *2012 IEEE International Conference on Services Computing (SCC)*, volume 00, pages 572–582, Nov. 2013. [cited at p. 56, 67]

[70] HPC team at Ghent University. Easybuild. https://github.com/hpcugent/easybuild, 03 2017. [cited at p. 56, 63, 67, 100]

[71] HPC team at Ghent University. A collection of easyconfig files that describe which software to build using which build options with EasyBuild. https://github.com/easybuilders/easybuild-easyconfigs, 09 2018. [cited at p. 167]

[72] Sascha Hunold and Jesper Larsson Träff. On the state and importance of reproducible experimental research in parallel computing. *CoRR*, abs/1308.3648, 2013. [cited at p. 50]

[73] Roberto Ierusalimschy. *Programming in Lua, Third Edition*. Lua.Org, 3rd edition, 2013. [cited at p. 43]

[74] John P. A. Ioannidis, David B. Allison, Catherine A. Ball, Issa Coulibaly, Xiangqin Cui, Aedin C. Culhane, Mario Falchi, Cesare Furlanello, Laurence Game, Giuseppe Jurman, Jon Mangion, Tapan Mehta, Michael Nitzberg, Grier P. Page, Enrico Petretto, and Vera van Noort. Repeatability of published microarray gene expression analyses. *Nat Genet*, 41(2):149–155, Feb 2009. [cited at p. 7, 8, 48, 49]

[75] B. R. Jasny, G. Chin, L. Chong, and S. Vignieri. Again, and again, and again ... *Science*, 334(6060):1225–1225, dec 2011. [cited at p. 8, 48]

[76] I. Jimenez, A. Arpaci-Dusseau, R. Arpaci-Dusseau, J. Lofstead, C. Maltzahn, K. Mohror, and R. Ricci. Popperci: Automated reproducibility validation. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 450–455, May 2017. [cited at p. 48, 50]

[77] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM. [cited at p. 38]

[78] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 51–60, New York, NY, USA, 2006. ACM. [cited at p. 97]

[79] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 37–37, New York, NY, USA, 2001. ACM. [cited at p. 87]

[80] Darren J. Kerbyson and Philip W. Jones. A performance model of the parallel ocean program. *The International Journal of High Performance Computing Applications*, 19(3):261–276, 2005. [cited at p. 87]

[81] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):e0177459, may 2017. [cited at p. 139]

[82] Randall J. LeVeque. Python tools for reproducible research on hyperbolic problems. *Computing in Science & Engineering*, 11(1):19–27, jan 2009. [cited at p. 7, 48]

[83] Mark Liberman. Replicability vs. reproducibility — or is it the other way around? http://languagelog.ldc.upenn.edu/nll/?p=21956, oct 2015. [cited at p. 4, 7, 48]

[84] Y.-M. Lin, C. Dimitrakopoulos, K. A. Jenkins, D. B. Farmer, H.-Y. Chiu, A. Grill, and Ph. Avouris. 100-GHz transistors from wafer-scale epitaxial graphene. *Science*, 327(5966):662–662, feb 2010. [cited at p. 18]

[85] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 129–148, Cham, 2015. Springer International Publishing. [cited at p. 91]

[86] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. [cited at p. 10, 48, 49]

[87] Antonio Maffia, Helmar Burkhart, and Danilo Guerrera. Reproducibility in practice: Lessons learned from research and teaching experiments. In *Euro-Par 2015: Parallel Processing Workshops*. Springer International Publishing, 2015. [cited at p. 10, 48, 50, 65, 121, 138]

[88] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck. *Crossroads*, 5(3es):2–es, apr 1999. [cited at p. 27]

[89] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing*, 37(4):C439–C464, jan 2015. [cited at p. 80]

[90] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002. [cited at p. 26]

[91] John Mccalpin and David Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report 379, Rutgers University, 1999. [cited at p. 97]

[92] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/. [cited at p. 70]

[93] Robert McLay. Lmod: Environmental modules system. http://www.tacc.utexas.edu/tacc-projects/lmod. [cited at p. 43]

[94] Jill P. Mesirov. Accessible reproducible research. *Science*, 327(5964):415–416, 2010. [cited at p. 8]

[95] Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Alexandra Nenadic, Ian Dunlop, Alan Williams, Tom Oinn, and Carole Goble. Taverna, reloaded. In Michael Gertz and Bertram Ludäscher, editors, *Scientific and Statistical Database Management*, volume 6187 of *Lecture Notes in Computer Science*, pages 471–481. Springer Berlin Heidelberg, 2010. [cited at p. 10, 48, 49]

[96] Gordon E. Moore. Cramming more components onto integrated circuits. *Integrated Circuits Electronics*, 38(8):114–117, April 1965. [cited at p. 15]

[97] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace–a toolset for the performance prediction of parallel and distributed systems. *The International Journal of High Performance Computing Applications*, 14(3):228–251, 2000. [cited at p. 87]

[98] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, jun 2004. [cited at p. 9, 49]

[99] OpenMP Architecture Review Board. The OpenMP® API specification for parallel programming. http://www.openmp.org, 2016. [cited at p. 38]

[100] Roger D. Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011. [cited at p. 8, 48]

[101] V. Petkov, M. Gerndt, and M. Firbach. Pathway: Performance analysis and tuning using workflows. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 792–799, Nov 2013. [cited at p. 11]

[102] Heather A. Piwowar, Roger S. Day, and Douglas B. Fridsma. Sharing detailed research data is associated with increased citation rate. *PLOS ONE*, 2(3):1–5, 03 2007. [cited at p. 9]

[103] Hans E. Plesser. Reproducibility vs. replicability: A brief history of a confused terminology. *Frontiers in Neuroinformatics*, 11:76, 2018. [cited at p. 138]

[104] Florian Prinz, Thomas Schlange, and Khusru Asadullah. Believe it or not: how much can we rely on published data on potential drug targets? *Nature Reviews Drug Discovery*, 10(9):712–712, aug 2011. [cited at p. 7, 8, 48, 49]

[105] Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011. [cited at p. 90]

[106] S.E. Raasch and S.K. Reinhardt. The impact of resource partitioning on SMT processors. In *Oceans 2002 Conference and Exhibition. Conference Proceedings (Cat. No.02CH37362)*. IEEE Comput. Soc. [cited at p. 27]

[107] Karthik Ram. Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine*, 8(1):7, 2013. [cited at p. 10]

[108] Michael Reich, Ted Liefeld, Joshua Gould, Jim Lerner, Pablo Tamayo, and Jill P. Mesirov. Genepattern 2.0. *Nat Genet*, 38(5):500–501, May 2006. [cited at p. 9, 48, 49]

[109] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. [cited at p. 38]

[110] Arch D. Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):66–71, mar 2013. [cited at p. 38]

[111] Rodrigo C. O. Rocha, Alyson D. Pereira, Luiz Ramos, and Luís F. W. Góes. Toast: Automatic tiling for iterative stencil computations on gpus. *Concurrency and Computation: Practice and Experience*, 29(8):e4053. [cited at p. 96]

[112] Thomas Roehl, Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid. `https://github.com/RRZE-HPC/likwid`, 08 2018. [cited at p. 31, 58, 68, 100]

[113] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Validation of hardware events for successful performance pattern identification in high performance computing. In Andreas Knüpfer, Tobias Hilbrich, Christoph Niethammer, José Gracia, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2015*, pages 17–28. Springer International Publishing, 2016. [cited at p. 58]

[114] Nicolas P. Rougier, Konrad Hinsen, Frédéric Alexandre, Thomas Arildsen, Lorena A. Barba, Fabien C.Y. Benureau, C. Titus Brown, Pierre de Buyl, Ozan Caglayan, Andrew P. Davison, Marc-André Delsuc, Georgios Detorakis, Alexandra K. Diem, Damien Drix, Pierre Enel, Benoît Girard, Olivia Guest, Matt G. Hall, Rafael N. Henriques, Xavier Hinaut, Kamil S. Jaron, Mehdi Khamassi, Almar Klein, Tiina Manninen, Pietro Marchesi, Daniel McGlinn, Christoph Metzner, Owen Petchey, Hans Ekkehard Plesser, Timothée Poisot, Karthik Ram, Yoav Ram, Etienne Roesch, Cyrille Rossant, Vahid Rostami, Aaron Shifman, Joseph Stachelek, Marcel Stimberg, Frank Stollmeier, Federico Vaggi, Guillaume Viejo, Julien Vitay, Anya E. Vostinar, Roman Yurchak, and Tiziano Zito. Sustainable computational science: the ReScience initiative. *PeerJ Computer Science*, 3:e142, dec 2017. [cited at p. 10, 138]

[115] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLOS Computational Biology*, 9(10):1–4, 10 2013. [cited at p. 9]

[116] N. Savage. First graphene integrated circuit. EEE Spectrum, Jun 2011. [cited at p. 18]

[117] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Claudio T. Silva. Querying and re-using workflows with VisTrails. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1251–1254, New York, NY, USA, 2008. ACM. [cited at p. 9, 10, 48, 49]

[118] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. Making scientific computations reproducible. *Computing in Science & Engineering*, 2(6):61–67, 2000. [cited at p. 9]

[119] Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. *IJHPCA*, 18(1):115–133, 2004. [cited at p. 97]

[120] Arfon M. Smith, Kyle E. Niemeyer, Daniel S. Katz, Lorena A. Barba, George Githinji, Melissa Gymrek, Kathryn D. Huff, Christopher R. Madan, Abigail Cabunoc Mayes, Kevin M. Moerman, Pjotr Prins, Karthik Ram, Ariel Rokem, Tracy K. Teal, Roman Valls Guimera, and Jacob T. Vanderplas. Journal of open source software (JOSS): design and first-year review. *PeerJ Computer Science*, 4:e147, feb 2018. [cited at p. 138]

[121] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *SIGPLAN Not.*, 34(5):215–228, may 1999. [cited at p. 97]

[122] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. *CoRR*, abs/1410.5010, 2014. [cited at p. 87, 89, 92]

[123] V. Stodden, M. McNutt, D. H. Bailey, E. Deelman, Y. Gil, B. Hanson, M. A. Heroux, J. P. A. Ioannidis, and M. Taufer. Enhancing reproducibility for computational methods. *Science*, 354(6317):1240–1241, dec 2016. [cited at p. 8]

[124] Victoria Stodden, Friederich Leisch, and Roger D. Peng, editors. *Implementing Reproducible Research*. The R Series. Chapman and Hall/CRC, 2014. [cited at p. 9]

[125] Victoria Stodden and Sheila Miguez. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *Journal of Open Research Software*, 2(1), jul 2014. [cited at p. 8]

[126] Victoria C. Stodden. Trust your science? open your data and code. *Amstat News*, 409:21–22, 2011. [cited at p. 7, 48]

[127] John Paul Strachan, Dmitri B Strukov, Julien Borghetti, J Joshua Yang, Gilberto Medeiros-Ribeiro, and R Stanley Williams. The switching location of a bipolar memristor: chemical, thermal and structural mapping. *Nanotechnology*, 22(25):254015, may 2011. [cited at p. 18]

[128] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH Comput. Archit. News*, 36(1):277–286, mar 2008. [cited at p. 87]

[129] Meropi Topalidou, Arthur Leblois, Thomas Boraud, and Nicolas P. Rougier. A long journey into reproducible computational neuroscience. *Frontiers in Computational Neuroscience*, 9, mar 2015. [cited at p. 10, 48]

[130] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010. [cited at p. 31, 58, 68]

[131] Jan Treibig and Georg Hager. Introducing a performance model for bandwidth-limited loop kernels. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, pages 615–624, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. [cited at p. 87]

[132] Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell, and John Shalf. Exasat: An exascale co-design tool for performance modeling. *The International Journal of High Performance Computing Applications*, 29(2):209–232, 2015. [cited at p. 90]

[133] Veronica J. Vieland. The replication requirement. *Nature Genetics*, 29(3):244–245, nov 2001. [cited at p. 8, 48]

[134] Gerhard Wellein. private communication. [cited at p. 77]

[135] Samuel Webb Williams, Andrew Waterman, and David A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008. [cited at p. 87, 89, 136]

[136] Frank L. H. Wolfs. Introduction to the scientific method. `http://teacher.nsrl.rochester.edu/phylabs/appendixe/appendixe.html`, 2013. [cited at p. 3]

[137] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 171–180, May 2000. [cited at p. 97]

Appendices

Appendix A

Stencils Source Code

A.1 2d-1r-iso-const-box

A.1.1 PATUS

```
1:  stencil 2d1risoconstbox (
2:    double grid U(0 .. M_MAX-1, 0 .. N_MAX-1),
3:    double param c0 = 0.2,
4:    double param c1 = 0.3,
5:    double param c2 = 0.65
6:  )
7:  {
8:    iterate while t < 200;
9:    domainsize = (1 .. M_MAX-2, 1 .. N_MAX-2);
10:
11:   initial
12:   {
13:      U[x, y; 0] = 1;
14:   }
15:
16:   operation
17:   {
18:     U[x, y; t+1] = c0 * U[x, y; t] +
19:                    c1 * (U[x, y - 1; t] + U[x - 1, y; t] +
20:                          U[x + 1, y; t] + U[x, y + 1; t]) +
21:                    c2 * (U[x - 1, y - 1; t] + U[x + 1, y - 1; t] +
22:                    U[x - 1, y + 1; t] + U[x + 1, y + 1; t])
23:     ;
24:   }
```

```
25: }
```

Listing A.1: *2-dimensional isotropic box constant stencil with radius 1 implemented in* PATUS

A.1.2 PLUTO

```
1:  #include <stdlib.h>
2:  #include <math.h>
3:
4:  #include "timing.h"
5:  #include "dim_input.h"
6:
7:  int main(int argc, char **argv)
8:  {
9:
10:   int M = M_MAX;
11:   int N = N_MAX;
12:   int repeat = T_MAX;
13:
14:   double ***a = malloc((sizeof(double**)*2));
15:   a[0] = (double**) malloc(sizeof(double*)*M);
16:   a[1] = (double**) malloc(sizeof(double*)*M);
17:   for(int i = 0; i < M; i++){
18:     a[0][i] = (double*) malloc(sizeof(double*)*N);
19:     a[1][i] = (double*) malloc(sizeof(double*)*N);
20:   }
21:
22:
23:   #pragma omp parallel for schedule(runtime)
24:   for (int j = 0; j < M; ++j)
25:     for (int i = 0; i < N; ++i)
26:       a[0][j][i] = rand() / ((double ) RAND_MAX);
27:
28:   timing(&wct_start, &cput_start);
29:   #pragma scop
30:   for (int n = 0; n < repeat; n++)
31:   {
32:     for(int j = 1; j < M - 1; j++){
33:       for(int i = 1; i < N - 1; i++){
34:         a[(n+1)%2][j][i] = c0 * a[n%2][j][i]
35:                 + c1 * (a[n%2][j][i-1] + a[n%2][j-1][i] +
36:                   a[n%2][j+1][i] + a[n%2][j][i+1])
37:                 + c2 * (a[n%2][j-1][i-1] + a[n%2][j+1][i-1] +
38:                   a[n%2][j-1][i+1] + a[n%2][j+1][i+1])
39:         ;
40:       }
```

```
41:       }
42:     }
43:     #pragma endscop
44:
45:     timing(&wct_end, &cput_end);
```

Listing A.2: *2-dimensional isotropic box constant stencil with radius 1 implemented in* PLUTO

A.1.3 OpenMP

```
1:  #include <stdlib.h>
2:  #include <math.h>
3:
4:  #include "timing.h"
5:  #include "kerncraft.h"
6:  #include "kernel.c"
7:
8:  #include "dim_input.h"
9:
10: void* aligned_malloc(size_t, size_t);
11: int main(int argc, char **argv)
12: {
13:
14:   int M = M_MAX;
15:   int N = N_MAX;
16:   int repeat = T_MAX;
17:
18:   double *a = aligned_malloc((sizeof(double)) * (M * N), 32);
19:   #pragma omp parallel for schedule(static)
20:   for (int i = 0; i < (M * N); ++i)
21:     a[i] = rand() / ((double ) RAND_MAX);
22:
23:   double *b = aligned_malloc((sizeof(double)) * (M * N), 32);
24:   #pragma omp parallel for schedule(static)
25:   for (int i = 0; i < (M * N); ++i)
26:     b[i] = rand() / ((double ) RAND_MAX);
27:
28:   ...
29:
30:   timing(&wct_start, &cput_start);
31:     for (int n = 0; n < repeat; ++n)
32:     {
33:       kernel_loop(a, b, c0, c1, c2, M, N);
34:       tmp = a;
35:       a = b;
36:       b = tmp;
```

```
37:       }
38:       timing(&wct_end, &cput_end);
39:
40:   }
41:
42:
43:   void kernel_loop(double *a, double *b, double c0, double c1,
44:     double c2, int M, int N)
45:   {
46:     #pragma omp parallel for schedule(static)
47:     for (int j = 1; j < (M - 1); j++)
48:     {
49:       for (int i = 1; i < (N - 1); i++)
50:       {
51:         b[i + (j * N)] = ((c0 * a[i + (j * N)]) + (
52:           c1 * (((a[(i - 1) + (j * N)] +
53:             a[i + ((j - 1) * N)]) +
54:             a[i + ((j + 1) * N)]) +
55:             a[(i + 1) + (j * N)]))) + (
56:           c2 * (((a[(i - 1) + ((j - 1) * N)] +
57:             a[(i - 1) + ((j + 1) * N)]) +
58:             a[(i + 1) + ((j - 1) * N)]) +
59:             a[(i + 1) + ((j + 1) * N)]));
60:       }
61:
62:     }
63:
64:   }
```

Listing A.3: *2-dimensional isotropic box constant stencil with radius 1 implemented in C + OpenMP*

A.2 2d-4r-iso-const-box

A.2.1 PATUS

```
1:  stencil 2d4risoconstbox (
2:    double grid U(0 .. M_MAX-1, 0 .. N_MAX-1),
3:    double param c0 = 0.2,
4:    double param c1 = 0.3,
5:    double param c2 = 0.65,
6:    double param c3 = 0.15,
7:    double param c4 = 0.75,
8:    double param c5 = 0.45,
9:    double param c6 = 0.25,
10:   double param c7 = 0.85,
```

```
11:    double param c8 = 0.35
12:    )
13:  {
14:    iterate while t < 200;
15:    domainsize = (4 .. M_MAX-5, 4 .. N_MAX-5);
16:
17:    initial
18:    {
19:       U[x, y; 0] = 1;
20:    }
21:
22:    operation
23:    {
24:      U[x, y; t+1] = c0 * U[x, y; t] +
25:                     c1 * (U[x, y - 1; t] + U[x - 1, y; t] +
26:                           U[x + 1, y; t] + U[x, y + 1; t]) +
27:                     c2 * (U[x, y - 2; t] + U[x - 1, y - 1; t] +
28:                           U[x + 1, y - 1; t] + U[x - 2, y; t] +
29:                           U[x + 2, y; t] + U[x - 1, y + 1; t] +
30:                           U[x + 1, y + 1; t] + U[x, y + 2; t]) +
31:                     c3 * (U[x, y - 3; t] + U[x - 1, y - 2; t] +
32:                           U[x + 1, y - 2; t] + U[x - 2, y - 1; t] +
33:                           U[x + 2, y - 1; t] + U[x - 3, y; t] +
34:                           U[x + 3, y; t] + U[x - 2, y + 1; t] +
35:                           U[x + 2, y + 1; t] + U[x - 1, y + 2; t] +
36:                           U[x + 1, y + 2; t] + U[x, y + 3; t]) +
37:                     c4 * (U[x, y - 4; t] + U[x - 1, y - 3; t] +
38:                           U[x + 1, y - 3; t] + U[x - 2, y - 2; t] +
39:                           U[x + 2, y - 2; t] + U[x - 3, y - 1; t] +
40:                           U[x + 3, y - 1; t] + U[x - 4, y; t] +
41:                           U[x + 4, y; t] + U[x - 3, y + 1; t] +
42:                           U[x + 3, y + 1; t] + U[x - 2, y + 2; t] +
43:                           U[x + 2, y + 2; t] + U[x - 1, y + 3; t] +
44:                           U[x + 1, y + 3; t] + U[x, y + 4; t]) +
45:                     c5 * (U[x - 1, y - 4; t] + U[x + 1, y - 4; t] +
46:                           U[x - 2, y - 3; t] + U[x + 2, y - 3; t] +
47:                           U[x - 3, y - 2; t] + U[x + 3, y - 2; t] +
48:                           U[x - 4, y - 1; t] + U[x + 4, y - 1; t] +
49:                           U[x - 4, y + 1; t] + U[x + 4, y + 1; t] +
50:                           U[x - 3, y + 2; t] + U[x + 3, y + 2; t] +
51:                           U[x - 2, y + 3; t] + U[x + 2, y + 3; t] +
52:                           U[x - 1, y + 4; t] + U[x + 1, y + 4; t]) +
53:                     c6 * (U[x - 2, y - 4; t] + U[x + 2, y - 4; t] +
54:                           U[x - 3, y - 3; t] + U[x + 3, y - 3; t] +
55:                           U[x - 4, y - 2; t] + U[x + 4, y - 2; t] +
56:                           U[x - 4, y + 2; t] + U[x + 4, y + 2; t] +
57:                           U[x - 3, y + 3; t] + U[x + 3, y + 3; t] +
```

```
58:                         U[x - 2, y + 4; t] + U[x + 2, y + 4; t]) +
59:                   c7 * (U[x - 3, y - 4; t] + U[x + 3, y - 4; t] +
60:                         U[x - 4, y - 3; t] + U[x + 4, y - 3; t] +
61:                         U[x - 4, y + 3; t] + U[x + 4, y + 3; t] +
62:                         U[x - 3, y + 4; t] + U[x + 3, y + 4; t]) +
63:                   c8 * (U[x - 4, y - 4; t] + U[x + 4, y - 4; t] +
64:                         U[x - 4, y + 4; t] + U[x + 4, y + 4; t])
65:     ;
66:   }
67: }
```

Listing A.4: *2-dimensional isotropic box constant stencil with radius 4 implemented in* PATUS

A.2.2 PLUTO

```
1:  #include <stdlib.h>
2:  #include <math.h>
3:
4:  #include "timing.h"
5:  #include "dim_input.h"
6:
7:  int main(int argc, char **argv)
8:  {
9:
10:   int M = M_MAX;
11:   int N = N_MAX;
12:   int repeat = T_MAX;
13:
14:   double ***a = malloc((sizeof(double**)*2));
15:   a[0] = (double**) malloc(sizeof(double*)*M);
16:   a[1] = (double**) malloc(sizeof(double*)*M);
17:   for(int i = 0; i < M; i++){
18:     a[0][i] = (double*) malloc(sizeof(double*)*N);
19:     a[1][i] = (double*) malloc(sizeof(double*)*N);
20:   }
21:
22:
23:   #pragma omp parallel for schedule(runtime)
24:   for (int j = 0; j < M; ++j)
25:     for (int i = 0; i < N; ++i)
26:       a[0][j][i] = rand() / ((double ) RAND_MAX);
27:
28:   timing(&wct_start, &cput_start);
29:   #pragma scop
30:   for (int n = 0; n < repeat; ++n)
31:   {
```

```
32:     for(int j=4; j < M-4; j++){
33:       for(int i=4; i < N-4; i++){
34:         a[(n+1)%2][j][i] = c0 * a[n%2][j][i]
35:           + c1 * (a[n%2][j][i-1] + a[n%2][j-1][i] +
36:             a[n%2][j+1][i] + a[n%2][j][i+1])
37:           + c2 * (
38:             a[n%2][j][i-2] + a[n%2][j-1][i-1] +
39:             a[n%2][j+1][i-1] + a[n%2][j-2][i] +
40:             a[n%2][j+2][i] + a[n%2][j-1][i+1] +
41:             a[n%2][j+1][i+1] + a[n%2][j][i+2])
42:           + c3 * (
43:             a[n%2][j][i-3] + a[n%2][j-1][i-2] +
44:             a[n%2][j+1][i-2] + a[n%2][j-2][i-1] +
45:             a[n%2][j+2][i-1] + a[n%2][j-3][i] +
46:             a[n%2][j+3][i] + a[n%2][j-2][i+1] +
47:             a[n%2][j+2][i+1] + a[n%2][j-1][i+2] +
48:             a[n%2][j+1][i+2] + a[n%2][j][i+3])
49:           + c4 * (
50:             a[n%2][j][i-4] + a[n%2][j-1][i-3] +
51:             a[n%2][j+1][i-3] + a[n%2][j-2][i-2] +
52:             a[n%2][j+2][i-2] + a[n%2][j-3][i-1] +
53:             a[n%2][j+3][i-1] + a[n%2][j-4][i] +
54:             a[n%2][j+4][i] + a[n%2][j-3][i+1] +
55:             a[n%2][j+3][i+1] + a[n%2][j-2][i+2] +
56:             a[n%2][j+2][i+2] + a[n%2][j-1][i+3] +
57:             a[n%2][j+1][i+3] + a[n%2][j][i+4])
58:           + c5 * (
59:             a[n%2][j-1][i-4] + a[n%2][j+1][i-4] +
60:             a[n%2][j-2][i-3] + a[n%2][j+2][i-3] +
61:             a[n%2][j-3][i-2] + a[n%2][j+3][i-2] +
62:             a[n%2][j-4][i-1] + a[n%2][j+4][i-1] +
63:             a[n%2][j-4][i+1] + a[n%2][j+4][i+1] +
64:             a[n%2][j-3][i+2] + a[n%2][j+3][i+2] +
65:             a[n%2][j-2][i+3] + a[n%2][j+2][i+3] +
66:             a[n%2][j-1][i+4] + a[n%2][j+1][i+4])
67:           + c6 * (
68:             a[n%2][j-2][i-4] + a[n%2][j+2][i-4] +
69:             a[n%2][j-3][i-3] + a[n%2][j+3][i-3] +
70:             a[n%2][j-4][i-2] + a[n%2][j+4][i-2] +
71:             a[n%2][j-4][i+2] + a[n%2][j+4][i+2] +
72:             a[n%2][j-3][i+3] + a[n%2][j+3][i+3] +
73:             a[n%2][j-2][i+4] + a[n%2][j+2][i+4])
74:           + c7 * (
75:             a[n%2][j-3][i-4] + a[n%2][j+3][i-4] +
76:             a[n%2][j-4][i-3] + a[n%2][j+4][i-3] +
77:             a[n%2][j-4][i+3] + a[n%2][j+4][i+3] +
78:             a[n%2][j-3][i+4] + a[n%2][j+3][i+4])
```

```
79:             + c8 * (
80:               a[n%2][j-4][i-4] + a[n%2][j+4][i-4] +
81:               a[n%2][j-4][i+4] + a[n%2][j+4][i+4]);
82:         }
83:       }
84:     }
85:     #pragma endscop
86:
87:     timing(&wct_end, &cput_end);
```

Listing A.5: *2-dimensional isotropic box constant stencil with radius 4 implemented in* PLUTO

A.2.3 OpenMP

```
1:  #include <stdlib.h>
2:  #include <math.h>
3:
4:  #include "timing.h"
5:  #include "kerncraft.h"
6:  #include "kernel.c"
7:
8:  #include "dim_input.h"
9:
10: void* aligned_malloc(size_t, size_t);
11: int main(int argc, char **argv)
12: {
13:
14:   int M = M_MAX;
15:   int N = N_MAX;
16:   int repeat = T_MAX;
17:
18:   double *a = aligned_malloc((sizeof(double)) * (M * N), 32);
19:   #pragma omp parallel for schedule(runtime)
20:   for (int i = 0; i < (M * N); ++i)
21:     a[i] = rand() / ((double ) RAND_MAX);
22:
23:   double *b = aligned_malloc((sizeof(double)) * (M * N), 32);
24:   #pragma omp parallel for schedule(runtime)
25:   for (int i = 0; i < (M * N); ++i)
26:     b[i] = rand() / ((double ) RAND_MAX);
27:
28:   ...
29:
30:   timing(&wct_start, &cput_start);
31:   for (int n = 0; n < repeat; ++n)
32:   {
```

```
33:     kernel_loop(a, b, c0, c1, c2, c3, c4, c5, c6, c7, c8, M, N);
34:     tmp = a;
35:     a = b;
36:     b = tmp;
37:   }
38:   timing(&wct_end, &cput_end);
39:
40: }
41:
42:
43: void kernel_loop(double *a, double *b, double c0, double c1,
44:   double c2, double c3, double c4, double c5, double c6,
45:   double c7, double c8, int M, int N)
46: {
47:   #pragma omp parallel for schedule(runtime)
48:   for (int j = 4; j < (M - 4); j++)
49:   {
50:     for (int i = 4; i < (N - 4); i++)
51:     {
52:       b[i + (j * N)] =
53:         c0 * a[i + (j * N)] +
54:         c1 * (
55:           a[(i - 1) + (j * N)] + a[i + ((j - 1) * N)] +
56:           a[i + ((j + 1) * N)] + a[(i + 1) + (j * N)]) +
57:         c2 * (
58:           a[(i - 2) + (j * N)] + a[(i - 1) + ((j - 1) * N)] +
59:           a[(i - 1) + ((j + 1) * N)] + a[i + ((j - 2) * N)] +
60:           a[i + ((j + 2) * N)] + a[(i + 1) + ((j - 1) * N)] +
61:           a[(i + 1) + ((j + 1) * N)] + a[(i + 2) + (j * N)]) +
62:         c3 * (
63:           a[(i - 3) + (j * N)] + a[(i - 2) + ((j - 1) * N)] +
64:           a[(i - 2) + ((j + 1) * N)] + a[(i - 1) + ((j - 2) * N)] +
65:           a[(i - 1) + ((j + 2) * N)] + a[i + ((j - 3) * N)] +
66:           a[i + ((j + 3) * N)] + a[(i + 1) + ((j - 2) * N)] +
67:           a[(i + 1) + ((j + 2) * N)] + a[(i + 2) + ((j - 1) * N)] +
68:           a[(i + 2) + ((j + 1) * N)] + a[(i + 3) + (j * N)])+
69:         c4 * (a[(i - 4) + (j * N)] +
70:           a[(i - 3) + ((j - 1) * N)] + a[(i - 3) + ((j + 1) * N)] +
71:           a[(i - 2) + ((j - 2) * N)] + a[(i - 2) + ((j + 2) * N)] +
72:           a[(i - 1) + ((j - 3) * N)] + a[(i - 1) + ((j + 3) * N)] +
73:           a[i + ((j - 4) * N)] + a[i + ((j + 4) * N)] +
74:           a[(i + 1) + ((j - 3) * N)] + a[(i + 1) + ((j + 3) * N)] +
75:           a[(i + 2) + ((j - 2) * N)] + a[(i + 2) + ((j + 2) * N)] +
76:           a[(i + 3) + ((j - 1) * N)] + a[(i + 3) + ((j + 1) * N)] +
77:           a[(i + 4) + (j * N)]) +
78:         c5 * (a[(i - 4) + ((j - 1) * N)] +
79:           a[(i - 4) + ((j + 1) * N)] + a[(i - 3) + ((j - 2) * N)] +
```

```
80:             a[(i − 3) + ((j + 2) ∗ N)] + a[(i − 2) + ((j − 3) ∗ N)] +
81:             a[(i − 2) + ((j + 3) ∗ N)] + a[(i − 1) + ((j − 4) ∗ N)] +
82:             a[(i − 1) + ((j + 4) ∗ N)] + a[(i + 1) + ((j − 4) ∗ N)] +
83:             a[(i + 1) + ((j + 4) ∗ N)] + a[(i + 2) + ((j − 3) ∗ N)] +
84:             a[(i + 2) + ((j + 3) ∗ N)] + a[(i + 3) + ((j − 2) ∗ N)] +
85:             a[(i + 3) + ((j + 2) ∗ N)] + a[(i + 4) + ((j − 1) ∗ N)] +
86:             a[(i + 4) + ((j + 1) ∗ N)]) +
87:           c6 ∗ (a[(i − 4) + ((j − 2) ∗ N)] +
88:             a[(i − 4) + ((j + 2) ∗ N)] + a[(i − 3) + ((j − 3) ∗ N)] +
89:             a[(i − 3) + ((j + 3) ∗ N)] + a[(i − 2) + ((j − 4) ∗ N)] +
90:             a[(i − 2) + ((j + 4) ∗ N)] + a[(i + 2) + ((j − 4) ∗ N)] +
91:             a[(i + 2) + ((j + 4) ∗ N)] + a[(i + 3) + ((j − 3) ∗ N)] +
92:             a[(i + 3) + ((j + 3) ∗ N)] + a[(i + 4) + ((j − 2) ∗ N)] +
93:             a[(i + 4) + ((j + 2) ∗ N)]) +
94:           c7 ∗ (a[(i − 4) + ((j − 3) ∗ N)] +
95:             a[(i − 4) + ((j + 3) ∗ N)] + a[(i − 3) + ((j − 4) ∗ N)] +
96:             a[(i − 3) + ((j + 4) ∗ N)] + a[(i + 3) + ((j − 4) ∗ N)] +
97:             a[(i + 3) + ((j + 4) ∗ N)] + a[(i + 4) + ((j − 3) ∗ N)] +
98:             a[(i + 4) + ((j + 3) ∗ N)]) +
99:           c8 ∗ (a[(i − 4) + ((j − 4) ∗ N)] +
100:            a[(i − 4) + ((j + 4) ∗ N)] + a[(i + 4) + ((j − 4) ∗ N)] +
101:          a[(i + 4) + ((j + 4) ∗ N)]);
102:      }
103:    }
104: }
```

Listing A.6: *2-dimensional isotropic box constant stencil with radius 4 implemented in C + OpenMP*

Appendix B

Creation of a MethodType

A *methodtype* contains all the information required to install its software stack and compile and run the source code that uses it. It is the connection ring between a method (as defined in theory in Section 9.3) and the software stack. Its main components are three scripts that describe the way parameters are set up, the compilation instructions and the command used to run it. A *methodType* relies on modules that should be installed via EasyBuild. EasyBuild offers installation recipes for a variety of scientific software packages: they are available upstream on their repository [71]. If a custom *easyconfig* is needed, then it must be part of the methodType folder.
One of the *methodTypes* used in the experiments part of this work is called PLUTO-pet-0.11. It is used to provide to the users of PROVA! the pet branch of the PLUTO stencil compiler. If one wants to create such a methodType, the content of the compile, run and setup_parameters scripts should be the one contained in Listings B.1, B.2, and B.3 respectively.
Since an easyconfig (i.e. an installation recipe for EasyBuild) is not available in the easyconfigs repository ([71]), it is necessary to provide it. In Listing B.4 is shown the content of PLUTO-pet-0.11.eb.

```
#!/bin/bash

module_home=`pwd`

#create folder where to put the binary files and the outputs
if [ ! -d bin ]; then
        mkdir bin
fi
if [ ! -d out ]; then
        mkdir out
fi

#to compile an openmp implementation you need only to call the makefile
cd src
#Compile source
make clean
make all

cd $module_home
```

Listing B.1: *Script used to compile a method whose methodType is PLUTO-pet-0.11*

```
#!/bin/bash

#Environment variable $CURRENT_NT contains the number of threads to use
#output file name passed as first argument

module_home=`pwd`

export OMP_NUM_THREADS=$CURRENT_NT

#LIKWID_CMD is exported in the environment by the main run script

if [ "$1" == "" ]; then
        echo "Output file not specified. Output on screen."
        $LIKWID_CMD $module_home/bin/project
else
        $LIKWID_CMD $module_home/bin/project >> $1
fi
```

Listing B.2: *Script used to run a method whose methodType is PLUTO-pet-0.11*

```
#!/bin/bash

#the environment PARAM_VALUES contains the values of the parameters
#project descriptor passed as first argument

#project descriptor
if [ "$1" == "" ]; then
        PD=../.project
else
        PD=$1
fi

#read parameters names and put into an array
read -ra PARAMS_NAME <<<"$(gawk -F '[""]' '/list/{print $4}' $PD)"

count=0

rm -rf src/dim_input.h

touch src/dim_input.h

echo "Compiling with parameters: $PARAM_VALUES"

for param in $PARAM_VALUES
do
        echo "#define ${PARAMS_NAME[$count]} $param" >> src/dim_input.h
        count=$((count + 1))
done
```

Listing B.3: *Script used to setup the parameters of a method whose methodType is PLUTO-pet-0.11*

```
# author: Danilo Guerrera
easyblock = 'ConfigureMake'

name = 'PLUTO-pet'
# this version of pluto is the development branch with pet. The tar has been
# produced after having typed the following commands:
# $ git clone git://repo.or.cz/pluto.git -b pet
version = '0.11.0'

homepage = 'http://pluto-compiler.sourceforge.net/'
description = """PLUTO is an automatic parallelization tool based on the
  polyhedral model. The polyhedral model for compiler optimization
  provides an abstraction to perform high-level transformations
  such as loop-nest optimization and parallelization on affine
  loop nests. Pluto transforms C programs from source to source
  for coarse-grained parallelism and data locality simultaneously.
  The core transformation framework mainly works by finding affine
  transformations for efficient tiling."""

toolchain = {'name': 'dummy', 'version': ''}

# compiler toolchain dependencies
comp=('GCC','4.9.3-2.25')
builddependencies = [
     comp,
     ('Clang', '3.4', '', comp),
     ('pkg-config', '0.29', '', comp),
     ('libyaml', '0.1.6', '', comp),
     ('Autotools', '20161011', '', True),
]

sources = [SOURCE_TAR_GZ]
#source_urls = ['http://sourceforge.net/projects/pluto-compiler/files/']
preconfigopts = "git submodule init && git submodule update &&
./apply_patches.sh && ./autogen.sh && "
#preconfigopts += 'CPPFLAGS="-I$EBROOTCLANG/include $CPPFLAGS"
  #LDFLAGS="-L$EBROOTCLANG/lib $LDFLAGS" '

configopts = "CC=clang CXX=clang++"
# --with-clang-prefix=$EBROOTCLANG --with-isl-prefix=$EBROOTISL"

buildininstalldir = True

runtest = "test"

sanity_check_paths = {
    'files': ["bin/polycc"],
    'dirs': [],
}

maxparallel = 4

moduleclass = 'devel'
```

Listing B.4: *Easyconfig of PLUTO-pet-0.11*

Appendix C

Walkthrough of STEMPEL: Kerncraft and PROVA! Interfaces

This Appendix describes how to use STEMPEL in combination with Kerncraft and PROVA!.

The first functionality of STEMPEL is the option to generate a pseudo-C code describing a stencil operator. The generator accepts command line parameters, specifying the characteristics of the stencil, as described in Sections 7.2. Listing C.1 shows a possible command to generate a 3-dimensional star stencil with isotropic and constant coefficients, and radius one. Its output, created in a format representing the input accepted by Kerncraft, is saved in a file called *stencil.c*, whose content is shown in Listing C.2.

```
$ stempel gen -D 3 -r 1 -k star -C constant \
    --isotropic --store stencil.c
```

Listing C.1: *Command used to create c.*

```
1:  double a[M][N][P];
2:  double b[M][N][P];
3:  double c0;
4:  double c1;
5:
6:  for(int k=1; k < M-1; k++){
7:      for(int j=1; j < N-1; j++){
8:          for(int i=1; i < P-1; i++){
9:              b[k][j][i] = c0 * a[k][j][i]
10:                 + c1 * ((
11:                     a[k][j][i-1] + a[k][j][i+1]) +
```

```
12:                         (a[k-1][j][i] + a[k+1][j][i]) +
13:                         (a[k][j-1][i] + a[k][j+1][i]));
14:             }
15:         }
16:     }
```

Listing C.2: *Pseudo-C code produced as an output by the STEMPEL stencil generator, accepting the command line shown in Listing C.1.*

The command shown in Listing C.3 requests the ECM prediction, using the layer condition cache simulator, on an Intel Xeon E5-2640 v4, the input size of the grid being 250^3, using one core.

```
$ kerncraft -P LC -p ECM stencil.c
    -m Intel_Xeon_CPU_E5-2640_v4.yml -D M 250
    -D N 250 -D P 250 --unit=FLOP/s --cores=1 -vv
```

Listing C.3: *Command issued to predict the performance of a stencil, with input grid size of 250^3, using the ECM model on an Intel Xeon E5-2640 v4.*

The Roofline model can be requested, for the same stencil and grid size, on the same machine, by using the command line presented in Listing C.4.

```
$ kerncraft -P LC -p Roofline stencil.c
    -m Intel_Xeon_CPU_E5-2640_v4.yml -D M 250
    -D N 250 -D P 250 --unit=FLOP/s --cores=10 -vv
```

Listing C.4: *Command issued to predict the performance of a stencil, with input grid size of 250^3, using the Roofline model on an Intel Xeon E5-2640 v4.*

After having predicted the performance through Kerncraft, it is possible to generate a benchmark code, starting from the previously created pseudo-code. A possible way of doing so is by issuing the command shown in Listing C.5.

```
$ stempel bench stencil.c -m Intel_Xeon_CPU_E5-2640_v4.yml \
    --store --nocli
```

Listing C.5: *Command used to generate a benchmark code for the pseudo-C code contained in stencil.c.*

To obtain a reproducible execution of an experiment, a tool such as PROVA! can help. As described in Section 5.2, the usage of PROVA! passes through the creation of a project, a method and its implementation first, as shown in Listing C.6.

```
$ source /export/hpwc/PROVA/util/BaseSetup.sh
$ workflow project -c -p stencil --params "X_MAX Y_MAX \
```

```
    Z_MAX" --values "100 100 100" --threads 2
$ workflow method -c -p stencil \
    -m OpenMP-4.5-GCC-7.3.0-2.30 -n openMP
$ workflow run_exp -p stencil -e 5 -d 250 250 250 \
    -m openMP -t 1 10 --pin node
```

Listing C.6: *Command used in* PROVA! *to create a project named stencil, having X_MAX Y_MAX Z_MAX as parameters and default values of 100.*

The sequence of commands shown in Listings C.2, C.3, C.4, C.5, C.6 can be substituted by a single command offered by STEMPEL and listed in Listing C.7. Such a command accepts the characteristics of the stencil to model, the descriptor of the architecture for which a prediction must be performed, paths to the home and a workspace used by PROVA!, together with parameters such as the number of executions and threads to use for the experiments, the *methodType*, paths to the likwid command and its library. The output of such a command is a folder containing: pseudo-C code, Kerncraft analysis (both ECM and Roofline model), reproducible performance results of the benchmark code. All the files mentioned above can be used for further analysis, such as the one presented in Section 10.2.1.

```
$ analysis -w $STEMPELWORKSPACE -k star -c isotropic \
    -C constant -r 1 -d 3 -m Intel_Xeon_CPU_E5-2640_v4.yml \
    -p $PROVAHOME $PROVAWORKSPACE  -e 5 -t 1 10 \
    --method_type OpenMP-4.5-GCC-7.3.0-2.30
    -l $LIKWID_INC $LIKWID_LIB --iaca --sizes 250 250 250
```

Listing C.7: *Command used to generate, model and execute a 3 dimensional star stencil with isotropic and constant coefficients, and radius one.*

List of Figures

2.1 Overview of the evolution of microprocessors: transistors count, clock frequency and power consumption. Note that the y-axis is presented in logarithmic scale and as relative value. Data taken from [1]. 16

2.2 Overview of the evolution of the gate length of the transistors used in microprocessors. Data taken from [4]. 17

2.3 Simplified schematization of a typical cache-based microprocessor. 20

2.4 Basic instruction cycle, broken into a series of stages, without pipelining. The light colors represent a single instruction, while the dark ones represent the active stage of the pipeline. . 22

2.5 Basic instruction cycle, broken into a series of stages, with pipelining. Each color represents a single instruction. 22

2.6 SIMD operation compared to a scalar operation over an array. 24

2.7 Superscalarity of instructions: multiple functional units allow to complete execute several instructions per cycle. Each color represents a single instruction broken into stages in the pipeline. 24

2.8 Flynn's taxonomy of parallel computers, as described in [40]. . 25

2.9 UMA system consisting of two dual-core chips. 29

2.10 ccNUMA system consisting of two NUMA domains and eight cores. 29

2.11 Performance of the wrongly (left) and correctly (right) initialized three dimensional isotropic constant stencil with radius 1. 32

2.12 Performance of a 2-dimensional isotropic box stencil with constant coefficients and radius 1, grid size of 10000^2 and NUMA unaware initialization. On the left the performance, varying the numbers iterations, on an Intel Xeon E5-2695 with Cluster on Die mode enabled and NUMA balancing disabled. On the right the performance of the same code, on the same machine, but with NUMA balancing enabled. 33

3.1 Amdhal's law [10]: parallel speedup vs sequential fraction, for ranges of the parallel fraction between 0.5 and 1. 36
3.2 Speedup as a function of the number of cores for the ranges of the parallel fraction 0.5 to 1 assuming Gustafson's law [60]. . . 37
3.3 Runtime performance of a 2-dimensional isotropic box stencil with constant coefficients, on a node of miniHPC (see Section 9.1.4), using explicit thread pinning. The input grid size is fixed to 3000^2, and 200 time-steps are executed in double precision. The difference in the performance of the two configurations is given by a different set of compilation flags used. 41

4.1 Space of Computational Experiments 51
4.2 Ecosystem for reproducible stencil experiments. 53

6.1 Architectural overview of PROVA!. 63
6.2 UML diagram schematizing the creation of a project in a PROVA! workspace. 64
6.3 UML diagram schematizing the creation of a method in a project. 64
6.4 UML diagram schematizing the creation of an experiment to run the methods created in a project. 65
6.5 UML diagram schematizing the creation of a graph showing the results of the execution of an experiment. 66
6.6 Performance graph of a 2D Gaussian blur, with naive OpenMP implementation both with and without explicit pinning on Mint, taken from [59]. The histogram shows the average value out of 5 executions, and the error bars the standard deviation. 68
6.7 Roofline for the Mint cluster with the three kernels implementing a 3D wave equation. The description of the experiment and the discussion of the results have been published in [59]. . 70

7.1 Arithmetic intensity in various computational codes. 76

7.2 2-dimensional star stencil with radius 1, constant and isotropic coefficients, yielding 6 FLOP. 77

7.3 3-dimensional star stencil with radius 1, constant and isotropic coefficients, yielding 8 FLOP. 77

7.4 2-dimensional star stencil with radius 2, constant and isotropic coefficients, yielding 11 FLOP. 78

7.5 2-dimensional star stencil with radius 3, constant and heterogeneous coefficients, yielding 25 FLOP. 78

7.6 3-dimensional box stencil with radius 1, variable and point-symmetric coefficients, yielding 40 FLOP. 79

7.7 Unravelling of a 3-dimensional, radius 1, box stencil with variable and point-symmetric coefficients in order to appreciate their symmetricity. 79

7.8 Architecture of STEMPEL: the generated stencil kernel is passed to Kerncraft, for the performance modeling, and to the benchmark generator, that interfaces with PROVA!. 81

8.1 The performance engineering cycle consists of three phases: analisys and prediction of the code's characteristics, reproducible experimentation and measurements of its run-time behavior, and optmization 86

8.2 Visualization of the Roofline model with explicit depiction of the compute bound area, delimited by the peak floating point performance P_{max}, and memory bound area, delimited by the applicable peak bandwidth b_s at a given arithmetic intensity I. 88

8.3 Visualization of the factors involved in the calculation of the ECM model: overlapping time of computations and store (T_{OL}), time for loading the data from L1 to the registers (T_{nOL}), time for loading the data from L2 to L1 (T_{L1-L2}), time for loading from L3 to L2 (T_{L2-L3}), and time for loading from memory to L3 (T_{L3-MEM}). 89

8.4 Overview of Kerncraft: the user provides kernel code, constants, and a machine descriptor. IACA or OSACA, pycachesim, and a compiler are employed to build the ECM and Roofline models. Figure adapted from 91

8.5 Jacobi algorithm for a 2D 5-point stencil update: in pink and red the points needed at time T_0 to obtain the point in yellow at time T_1. The shadow indicates the points that are needed for the actual computation: if at least two successive rows can be kept in the cache, only one cell per update has to be fetched from memory (in red). 95

10.1 Percentage error of the prediction obtained through ECM and Roofline model, applied to a 2-dimensional star stencil with homogeneous, heterogeneous, point-symmetric and isotropic constant coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through PROVA!. 113

10.2 Percentage error of the prediction obtained through ECM and Roofline model, applied to a 2-dimensional star stencil with homogeneous, heterogeneous, point-symmetric and isotropic variable coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through PROVA!. 114

10.3 Percentage error of the prediction obtained through ECM and Roofline model, applied to a 2-dimensional box stencil with homogeneous, heterogeneous, point-symmetric and isotropic constant coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through PROVA!. 115

10.4 Percentage error of the prediction obtained through ECM and Roofline model, applied to a 2-dimensional box stencil with homogeneous, heterogeneous, point-symmetric and isotropic variable coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through PROVA!. 116

10.5 Percentage error of the prediction obtained through ECM and Roofline model, applied to a 3-dimensional star stencil with homogeneous, heterogeneous, point-symmetric and isotropic constant coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through PROVA!. 117
10.6 Percentage error of the prediction obtained through ECM and Roofline model, applied to a 3-dimensional star stencil with homogeneous, heterogeneous, point-symmetric and isotropic variable coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through PROVA!. 118
10.7 Percentage error of the prediction obtained through ECM and Roofline model, applied to a 3-dimensional box stencil with homogeneous, heterogeneous, point-symmetric and isotropic constant coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through PROVA!. 119
10.8 Percentage error of the prediction obtained through ECM and Roofline model, applied to a 3-dimensional box stencil with homogeneous, heterogeneous, point-symmetric and isotropic variable coefficients, on a socket (10 cores) of Emmy (left) and miniHPC (right), against the measured performance. The predicted performance has been obtained via Kerncraft, and the measured performance through PROVA!. 120
10.9 Performance graph of a 2D isotropic, box stencil with constant coefficients and radius 1. Two dimensions for the grids have been used: 500^2 and 3000^2, while the timesteps are 200. On the left are presented the results obtained on Emmy, on the right the ones on miniHPC. The kernel has been implemented using three methods: naive OpenMP implementation with explicit pinning using the strategies ByNode and BySpreading, PATUS without pinning, and PLUTO with explicit pinning using the strategies ByNode and BySpreading. For all the graphs the histogram shows the average value out of 10 executions, and the error bars the standard deviation. 128

10.10Performance graph of a 2D isotropic, box stencil with constant coefficients and radius 4. Two dimensions for the grids have been used: 500^2 and 3000^2, while the timesteps are 200. On the left are presented the results obtained on Emmy, on the right the ones on miniHPC. The kernel has been implemented using three methods: naive OpenMP implementation with explicit pinning using the strategies ByNode and BySpreading, PATUS without pinning, and PLUTO with explicit pinning using the strategies ByNode and BySpreading. For all the graphs the histogram shows the average value out of 10 executions, and the error bars the standard deviation. 129

List of Tables

4.1 Timeline of the interest in reproducibility and leading publications. 48

8.1 Timeline of performance modeling research and the most relevant publications. 87

9.1 Table representing all the combinations of the characteristic parameters of a star stencil, depicting the values used for the validation experiments. 105

9.2 Table representing all the combinations of the characteristic parameters of a box stencil, depicting the values used for the validation experiments. 106

Listings

2.1 NUMA unaware initialization of the arrays used to calculate a a three-dimensional isotropic star stencil with radius 1 30

2.2 NUMA aware initialization of the arrays used to calculate a a three-dimensional isotropic star stencil with radius 1 . 31

5.1 A possible command in PROVA! to run an experiment involving a 2-dimensional isotropic constant star stencil with grid size of 3000^2 using several thread counts and core affinity by node. 55

5.2 Command used in PROVA! to create a project named KNL having X_MAX, Y_MAX, Z_MAX as parameters and default values of 100. 58

5.3 Command used in PROVA! to create an implementation named wave (of type OpenMP-4.0-GCC-4.9.3-2.25) and belonging to the project KNL. 59

5.4 Command used in PROVA! to compile the implementation named wave and belonging to the project KNL. 59

5.5 Command used in PROVA! to execute with default parameters and no explicit pinning the implementation named wave and belonging to the project KNL. 60

5.6 Command used in PROVA! to execute an experiment with non-default parameters and no explicit pinning. A list of of threads to use and input parameters is passed on the command line. 60

5.7 Command used in PROVA! to execute an experiment with non-default parameters and explicit pinning by node. The experiment is submitted through the job scheduler PBS. A list of of threads to use and input parameters is passed on the command line. 60
5.8 Command used in PROVA! to generate an histogram of the performance output obtained in the experiment named 20171129_140354. Additional parameters restrict the data to be visualized by specifying the number of threads, the metric to visualize and the error bar. 61
6.1 Example of how the pinning strategies defined by PROVA! translate into a likwid command. 69
9.1 Topology of a node of Emmy, obtained via likwid-topology 103
9.2 Topology of a node of MiniHPC, obtained using the utility likwid-topology 104
9.3 Kernel of a 2-dimensional radius 1 isotropic box stencil with constant coefficients 107
9.4 Kernel of a 2-dimensional radius 4 isotropic box stencil with constant coefficients 108
10.1 A possible command used to obtain the performance prediction of Roofline and ECM models for a stencil on an Ivy Bridge E5-2660v2 machine. 121
10.2 ECM prediction through Kerncrat of a 2-dimensional, radius 4, isotropic box stencil with constant coefficients and size 3000^2 on Emmy. 124
10.3 ECM prediction through Kerncrat of a 2-dimensional, radius 4, isotropic box stencil with constant coefficients and size 3000^2 on miniHPC. 125
10.4 ECM prediction through Kerncrat of a 2-dimensional, radius 4, isotropic box stencil with constant coefficients and size 500^2 on Emmy. 125
10.5 ECM prediction through Kerncrat of a 2-dimensional, radius 4, isotropic box stencil with constant coefficients and size 500^2 on miniHPC 126
A.1 2-dimensional isotropic box constant stencil with radius 1 implemented in PATUS 157
A.2 2-dimensional isotropic box constant stencil with radius 1 implemented in PLUTO 158

A.3 2-dimensional isotropic box constant stencil with radius 1 implemented in C + OpenMP . . . 159
A.4 2-dimensional isotropic box constant stencil with radius 4 implemented in PATUS . . . 160
A.5 2-dimensional isotropic box constant stencil with radius 4 implemented in PLUTO . . . 162
A.6 2-dimensional isotropic box constant stencil with radius 4 implemented in C + OpenMP . . . 164
B.1 Script used to compile a method whose methodType is PLUTO-pet-0.11 . . . 168
B.2 Script used to run a method whose methodType is PLUTO-pet-0.11 . . . 168
B.3 Script used to setup the parameters of a method whose methodType is PLUTO-pet-0.11 . . . 169
B.4 Easyconfig of PLUTO-pet-0.11 . . . 170
C.1 Command used to create c. . . . 171
C.2 Pseudo-C code produced as an output by the STEMPEL stencil generator, accepting the command line shown in Listing C.1. . . . 171
C.3 Command issued to predict the performance of a stencil, with input grid size of 250^3, using the ECM model on an Intel Xeon E5-2640 v4. . . . 172
C.4 Command issued to predict the performance of a stencil, with input grid size of 250^3, using the Roofline model on an Intel Xeon E5-2640 v4. . . . 172
C.5 Command used to generate a benchmark code for the pseudo-C code contained in stencil.c. . . . 172
C.6 Command used in PROVA! to create a project named stencil, having X_MAX Y_MAX Z_MAX as parameters and default values of 100. . . . 172
C.7 Command used to generate, model and execute a 3 dimensional star stencil with isotropic and constant coefficients, and radius one. . . . 173

Curriculum Vitae

Personal Data

Name	Danilo Guerrera
Data of Birth	11.12.1986
Place of Birth	Benevento, Italy
Parents	Mario Guerrera and Carmelina Corbo
Nationality	Italian
Address	Andlauerstrasse 2, 4057 Basel

Education

1999 - 2004	Liceo Scientifico "G. Rummo" in Benevento (Italy)
2004 - 2012	M.Sc. and B.Sc. in Computer Science Engineering at the University of Sannio, Benevento (Italy)
2013 - 2018	Ph.D. candidate in Computer Science at the University of Basel (Switzerland)
Jun. - Dec. 2017	Scientific Researcher at the Friederich-Alexander University of Erlangen-Nürnberg (Germany) by the Regionale RechenZentrum Erlangen (RRZE)
20 Nov. 2018	Ph.D. examination, University of Basel (Switzerland)